PRATHYUSHA

ENGINEERING COLLEGE

DEPARTMENT OF CSE

II YEAR – IV SEMESTER

REGULATION 2021

CS3401- ALGORITHMS

CS3401   ALGORITHMS

COURSE OBJECTIVES:
• To understand and apply the algorithm analysis techniques on searching and sorting algorithms
• To critically analyze the efficiency of graph algorithms
• To understand different algorithm design techniques
• To solve programming problems using state space tree
• To understand the concepts behind NP Completeness, Approximation algorithms and randomized algorithms.

UNIT I   INTRODUCTION

Algorithm analysis: Time and space complexity - Asymptotic Notations and its properties Best case, Worst case and average case analysis – Recurrence relation: substitution method - Lower bounds – searching: linear search, binary search and Interpolation Search, Pattern search: The naïve string- matching algorithm - Rabin-Karp algorithm - Knuth-Morris-Pratt algorithm. Sorting: Insertion sort – heap sort

UNIT II GRAPH ALGORITHMS

Graph algorithms: Representations of graphs - Graph traversal: DFS – BFS - applications - Connectivity, strong connectivity, bi-connectivity - Minimum spanning tree: Kruskal's and Prim's algorithm- Shortest path: Bellman-Ford algorithm - Dijkstra's algorithm - Floyd-Warshall algorithm Network flow: Flow networks - Ford-Fulkerson method – Matching: Maximum bipartite matching

UNIT III ALGORITHM DESIGN TECHNIQUES

Divide and Conquer methodology: Finding maximum and minimum - Merge sort - Quick sort Dynamic programming: Elements of dynamic programming — Matrix-chain multiplication - Multi stage graph — Optimal Binary Search Trees. Greedy Technique: Elements of the greedy strategy - Activity-selection problem –- Optimal Merge pattern — Huffman Trees.

UNIT IV STATE SPACE SEARCH ALGORITHMS

Backtracking: n-Queens problem - Hamiltonian Circuit Problem - Subset Sum Problem – Graph colouring problem Branch and Bound: Solving 15-Puzzle problem - Assignment problem - Knapsack Problem - Travelling Salesman Problem

UNIT V NP-COMPLETE AND APPROXIMATION ALGORITHM

Tractable and intractable problems: Polynomial time algorithms – Venn diagram representation - NP- algorithms - NP-hardness and NP-completeness – Bin Packing problem - Problem reduction: TSP – 3- CNF problem. Approximation Algorithms: TSP - Randomized Algorithms: concept and application - primality testing - randomized quick sort - Finding kth smallest number

45 PERIODS

PRACTICAL EXERCISES: 30 PERIODS

Searching and Sorting Algorithms
1. Implement Linear Search. Determine the time required to search for an element. Repeat the experiment for different values of n, the number of elements in the list to be searched and plot a graph of the time taken versus n.
2. Implement recursive Binary Search. Determine the time required to search an element. Repeat the experiment for different values of n, the number of elements in the list to be searched and plot a graph of the time taken versus n.

3. Given a text txt [0...n-1] and a pattern pat [0...m-1], write a function search (char pat [ ], char txt [ ]) that prints all occurrences of pat [ ] in txt [ ]. You may assume that n > m.
4. Sort a given set of elements using the Insertion sort and Heap sort methods and determine the time required to sort the elements. Repeat the experiment for different values of n, the number of elements in the list to be sorted and plot a graph of the time taken versus n.

Graph Algorithms
1. Develop a program to implement graph traversal using Breadth First Search
2. Develop a program to implement graph traversal using Depth First Search
3. From a given vertex in a weighted connected graph, develop a program to find the shortest paths to other vertices using Dijkstra's algorithm.
4. Find the minimum cost spanning tree of a given undirected graph using Prim's algorithm.
5. Implement Floyd's algorithm for the All-Pairs- Shortest-Paths problem.
6. Compute the transitive closure of a given directed graph using Warshall's algorithm.

Algorithm Design Techniques
1. Develop a program to find out the maximum and minimum numbers in a given list of n numbers using the divide and conquer technique.
2. Implement Merge sort and Quick sort methods to sort an array of elements and determine the time required to sort. Repeat the experiment for different values of n, the number of elements in the list to be sorted and plot a graph of the time taken versus n.

State Space Search Algorithms
1. Implement N Queens problem using Backtracking.

Approximation Algorithms Randomized Algorithms
1. Implement any scheme to find the optimal solution for the Traveling Salesperson problem and then solve the same problem instance using any approximation algorithm and determine the error in the approximation.
2. Implement randomized algorithms for finding the kth smallest number. The programs can be implemented in C/C++/JAVA/ Python.

COURSE OUTCOMES: At the end of this course, the students will be able to:
CO1: Analyze the efficiency of algorithms using various frameworks
CO2: Apply graph algorithms to solve problems and analyze their efficiency.
CO3: Make use of algorithm design techniques like divide and conquer, dynamic programming and greedy techniques to solve problems
CO4: Use the state space tree method for solving problems.
CO5: Solve problems using approximation algorithms and randomized algorithms

TEXT BOOKS:

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, "Introduction to Algorithms", 3rd Edition, Prentice Hall of India, 2009.
2. Ellis Horowitz, Sartaj Sahni, Sanguthevar Rajasekaran "Computer Algorithms/C++" Orient Blackswan, 2nd Edition, 2019.

REFERENCES:

1. Anany Levitin, "Introduction to the Design and Analysis of Algorithms", 3rd Edition, Pearson Education, 2012.
2. Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman, "Data Structures and Algorithms", Reprint Edition, Pearson Education, 2006.
3. S. Sridhar, "Design and Analysis of Algorithms", Oxford university press, 2014.

Algorithm analysis: Time and space complexity - Asymptotic Notations and its properties Best case, Worst case and average case analysis – Recurrence relation: substitution method - Lower bounds – **Searching:** linear search, binary search and Interpolation Search, **Pattern search:** The naïve string-matching algorithm - Rabin-Karp algorithm - Knuth-Morris-Pratt algorithm. **Sorting:** Insertion sort – heap sort

Definition: Algorithm

An algorithm is a sequence of unambiguous instruction for solving a problem, for obtaining a required output for any legitimate input in a finite amount of time.

"Algorithmic is more than the branch of computer science. It is the core of computer science, and, in all fairness, can be said to be relevant it most of science, business and technology"

Characteristics of an algorithm:

i. **Non – ambiguity / Precise :** Each step in an algorithm should be non- ambiguous. i.e each instruction should be clear and precise.
ii. **Finiteness:** The algorithm should be finite. The algorithm must be terminated in a specified time.
iii. **Uniqueness:** The algorithm must be specified a required output.
iv. **Input:** Algorithm receives input
v. **Output:** Algorithm produces output
vi. **Generality:** The algorithm must works for all set of inputs.

Notion of an Algorithm

An algorithm is a sequence of unambiguous instruction for solving a problem, for obtaining a required output for any legitimate input in a finite amount of time.

➢ The nonambiguity requirement for each step of an algorithm cannot be compromised.
➢ The range of inputs for which an algorithm works has to be specified carefully.
➢ The same algorithm can be represented in several different ways.
➢ There may exist several algorithms for solving the same problem.
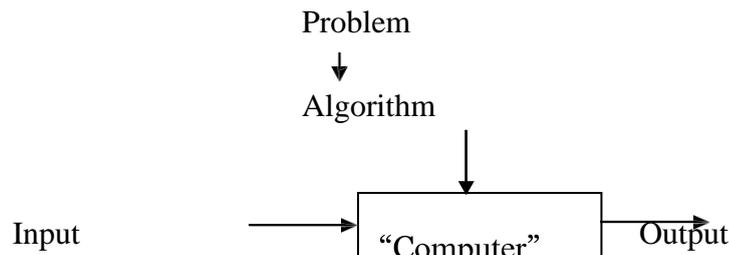
1) Consecutive integer checking algorithm for computing gcd(m, n)

Step 1 Assign the value of min{m, n} to t.
Step 2 Divide m by t. If the remainder of this division is 0, go to Step 3; otherwise, go to Step 4.
Step 3 Divide n by t. If the remainder of this division is 0, return the value of t as the answer and stop; otherwise, proceed to Step 4.
Step 4 Decrease the value of t by 1. Go to Step 2.

2) Middle-school procedure for computing gcd(m, n)

Step 1 Find the prime factors of m.
Step 2 Find the prime factors of n.

Step 3 Identify all the common factors in the two prime expansions found in Step 1 and Step 2.
Step 4 Compute the product of all the common factors and return it as the greatest common divisor of the numbers given.
Thus, for the numbers 60 and 24,

we get $\quad$ 60 = 2 . 2 . 3 . 5
$\qquad$ 24 = 2 . 2 . 2 . 3
$\quad$ gcd(60, 24) = 2 . 2 . 3 = 12

3) Euclid's algorithm for computing gcd(m, n)

Step 1 If n = 0, return the value of m as the answer and stop; otherwise, proceed to Step 2.
Step 2 Divide m by n and assign the value of the remainder to r.
Step 3 Assign the value of n to m and the value of r to n. Go to Step 1.

PSEUDOCODE:
ALGORITHM Euclid(m, n)
//Computes gcd(m, n) by Euclid's algorithm
//Input: Two nonnegative, not-both-zero integers m and n
//Output: Greatest common divisor of m and n
while n = 0 do
r ← m mod n
m ← n
n ← r
return m

ASYMPTOTIC NOTAIONS

Asymptotic notations are mathematical tools used to analyze the algorithm in terms of time efficiency.

Types Asymptotic Notations:
1. O Notation (Big Oh)
2. Ω Notation (Big Omega)
3. θ Notation (Big Theta)

1. O Notation (Big Oh):

Definition: A function f(n) is said to be in O(g(n)), denoted f(n) ∈ O(g(n)), if f(n) is bounded above by some constant multiple of g(n) for all large n, i.e., if there exist some positive constant c and some nonnegative integer n0 such that
f(n) ≤ cg(n) for all n ≥ n0 and c > 0.
O notation analyses the worst case for the given function.
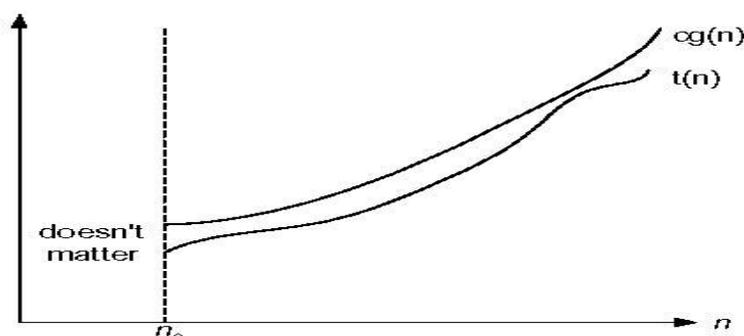The definition is illustrated in the following figure.

Figure 2.1 Big-oh notation: $t(n) \in O(g(n))$

Here n is size of an input &f(n) is a function of n.
When the input size increases, then the time is also increases.

Example:

Let take f(n)=3n+2 and g(n)=n.

We have to prove f(n) $\in O(g(n))$. By the definition,

f(n)$\leq$c g(n)

$3n+2 \leq c * n$ where $c>0$, $n_0 \geq 1$.

We can substitute any value for c. The best option is,

When c=4, $3n+2 \leq 4 * n$.

Most of the cases, 4*n is greater than 3n+2. Where $n \geq 2$.

Reason for taking n≥2:

If n=1, $3(1)+2 \leq 4(1) \Rightarrow 5 \leq 4 \Rightarrow$ Becomes False.

If n=2, $3(2)+2 \leq 4(2) \Rightarrow 8 \leq 8 \Rightarrow$ Becomes True.

If n=3, $3(3)+2 \leq 4(3) \Rightarrow 11 \leq 12 \Rightarrow$ Becomes True. If n=4, $3(4)+2 \leq 4(4) \Rightarrow 14 \leq 16 \Rightarrow$ Becomes True. And so on.

Therefore **$3n+2 \in O(n)$**.

2. Ω Notation (Big Omega):

Definition: A function f(n) is said to be in $\Omega(g(n))$, denoted f(n) $\in \Omega(g(n))$, if f(n) is bounded below by some positive constant multiple of g(n) for all large n, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

f(n) $\geq$ cg(n) for all n $\geq n_0$ and c > 0.

Ω notation analyses the **best case** for the given function.
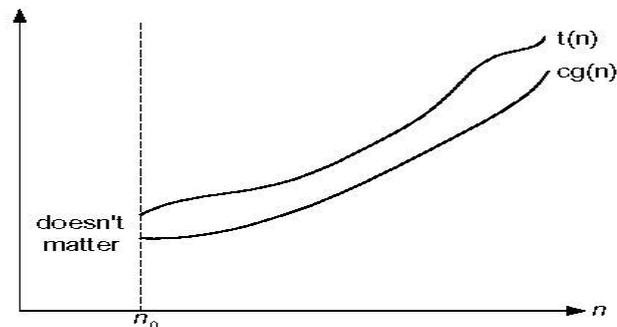
The definition is illustrated in the following figure.

Fig. 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$

If n=1, $3(1)+2 \geq 1(1) \Rightarrow 5 \geq 1 \Rightarrow$ Becomes True.

If n=2, $3(2)+2 \geq 1(2) \Rightarrow 8 \geq 2 \Rightarrow$ Becomes True.

If n=3, $3(3)+2 \geq 1(3) \Rightarrow 11 \geq 3 \Rightarrow$ Becomes True. And so on.

Therefore **$3n+2 \in \Omega(n)$**.

3. θ Notation (Big Theta):

Definition: A function f(n) is said to be in $\theta(g(n))$, denoted f(n) $\in \theta(g(n))$, if f(n) is bounded both above and below by some positive constant multiples of g(n) for all large n, i.e., if there exist some positive constants c1 and c2 and some nonnegative integer n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all n $\geq n_0$. θ notation analyses the **average case** for the given function.
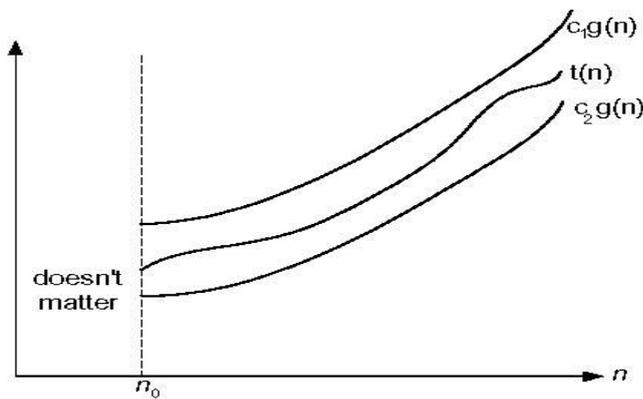
Figure 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$

Here n is size of an input &f(n) is a function of n.

When the input size increases, then the time is also increases. c_1 and c_2 are different constants.

f(n) is bounded by both upper and lower i.e) $c_1g(n) \le f(n) \le c_2g(n)$.

Example:

Let take f(n)=3n+2 and g(n)=n.

We have to prove f(n) $\in\theta$(g(n)). By the definition,

$c_1g(n) \le f(n) \le c_2g(n)$

The definition is illustrated in the following figure.

$c_1 * n \le 3n+2 \le c_2 * n$ $3n+2 \le c_2 * n$ when

c_2=4and 3n+2$\ge c_1 * n$ when c_1=1.

Such that, $1*n \le 3n+2 \le 4*n$. Where n$\ge$2. Therefore **3n+2**

=θ(n).

Useful Property Involving the Asymptotic Notations

Property 1:

If t1(n) $\in$ O(g1(n)) and t2(n) $\in$ O(g2(n)), then t1(n) + t2(n) $\in$ O(max{g1(n), g2(n)}).

PROOF:

The proof extends to orders of growth the following simple fact about four arbitrary real numbers a1, b1, a2, b2: if a1 $\le$ b1 and a2 $\le$ b2, then a1 + a2 $\le$ 2 max{b1, b2}.

Since t1(n) $\in$ O(g1(n)), there exist some positive constant c1 and some nonnegative integer n1 such that

t1(n) $\le$ c1g1(n) for all n $\ge$ n1.

Similarly, since t2(n) $\in$ O(g2(n)),

t2(n) $\le$ c2g2(n) for all n $\ge$ n2.

Let us denote c3 = max{c1, c2} and consider n $\ge$ max{n1, n2} so that we can use both inequalities. Adding them yields the following:

t1(n) + t2(n) $\le$ c1g1(n) + c2g2(n)

$\le$ c3g1(n) + c3g2(n) = c3[g1(n) + g2(n)] $\le$ c32 max{g1(n), g2(n)}.

Hence, t1(n) + t2(n) $\in$ O(max{g1(n), g2(n)}), with the constants c and n0 required by the O definition being 2c3 = 2 max{c1, c2} and max{n1, n2}, respectively.

Limits for Comparing Orders of Growth

$$\lim_{n \to \infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & \text{implies that } t(n) \text{ has a smaller order of growth than } g(n), \\ c & \text{implies that } t(n) \text{ has the same order of growth as } g(n), \\ \infty & \text{implies that } t(n) \text{ has a larger order of growth than } g(n).[3] \end{cases}$$

L'Hopital's rule

$$\lim_{n \to \infty} \frac{t(n)}{g(n)} = \lim_{n \to \infty} \frac{t'(n)}{g'(n)}$$

and Stirling's formula

$$n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \quad \text{for large values of } n.$$

EXAMPLE 1 Compare the orders of growth of $\frac{1}{2}n(n-1)$ and n^2. (This is one of the examples we used at the beginning of this section to illustrate the definitions.)

$$\lim_{n \to \infty} \frac{\frac{1}{2}n(n-1)}{n^2} = \frac{1}{2} \lim_{n \to \infty} \frac{n^2 - n}{n^2} = \frac{1}{2} \lim_{n \to \infty} (1 - \frac{1}{n}) = \frac{1}{2}.$$

Since the limit is equal to a positive constant, the functions have the same order of growth or, symbolically, $\frac{1}{2}n(n-1) \in \Theta(n^2)$. ■

EXAMPLE 2 Compare the orders of growth of $\log_2 n$ and $\sqrt{n}$. (Unlike Example 1, the answer here is not immediately obvious.)

$$\lim_{n \to \infty} \frac{\log_2 n}{\sqrt{n}} = \lim_{n \to \infty} \frac{(\log_2 n)'}{(\sqrt{n})'} = \lim_{n \to \infty} \frac{(\log_2 e) \frac{1}{n}}{\frac{1}{2\sqrt{n}}} = 2 \log_2 e \lim_{n \to \infty} \frac{1}{\sqrt{n}} = 0.$$

Since the limit is equal to zero, $\log_2 n$ has a smaller order of growth than $\sqrt{n}$. (Since $\lim_{n \to \infty} \frac{\log_2 n}{\sqrt{n}} = 0$, we can use the so-called *little-oh notation*: $\log_2 n \in o(\sqrt{n})$.)

EXAMPLE 3 Compare the orders of growth of $n!$ and 2^n. (We discussed this informally in Section 2.1.) Taking advantage of Stirling's formula, we get

$$\lim_{n \to \infty} \frac{n!}{2^n} = \lim_{n \to \infty} \frac{\sqrt{2\pi n} \left(\frac{n}{e}\right)^n}{2^n} = \lim_{n \to \infty} \sqrt{2\pi n} \frac{n^n}{2^n e^n} = \lim_{n \to \infty} \sqrt{2\pi n} \left(\frac{n}{2e}\right)^n = \infty.$$

Thus, though 2^n grows very fast, $n!$ grows still faster. We can write symbolically that $n! \in \Omega(2^n)$; note, however, that while the big-Omega notation does not preclude the possibility that $n!$ and 2^n have the same order of growth, the limit computed here certainly does. ■

Basic Efficiency Classes

Class	Name	Comments
1	*constant*	Short of best–case efficiencies, very few reasonable examples can be given since an algorithm's running time typically goes to infinity when its input size grows infinitely large.
$\log n$	*logarithmic*	Typically, a result of cutting a problem's size by a constant factor on each iteration of the algorithm (see Section 4.4). Note that a logarithmic algorithm cannot take into account all its input or even a fixed fraction of it: any algorithm that does so will have at least linear running time.
n	*linear*	Algorithms that scan a list of size n (e.g., sequential search) belong to this class.
$n \log n$	*linearithmic*	Many divide-and-conquer algorithms (see Chapter 5), including mergesort and quicksort in the average case, fall into this category.
n^2	*quadratic*	Typically, characterizes efficiency of algorithms with two embedded loops (see the next section). Elementary sorting algorithms and certain operations on $n \times n$ matrices are standard examples.
n^3	*cubic*	Typically, characterizes efficiency of algorithms with three embedded loops (see the next section). Several nontrivial algorithms from linear algebra fall into this class.
2^n	*exponential*	Typical for algorithms that generate all subsets of an n-element set. Often, the term "exponential" is used in a broader sense to include this and larger orders of growth as well.
$n!$	*factorial*	Typical for algorithms that generate all permutations of an n-element set.

ALGORITHM ANALYSIS:

♦ There are two kinds of efficiency
    ♦ **Time efficiency** - indicates how fast an algorithm in question runs.
    ♦ **Space efficiency** - deals with the extra space the algorithm requires.

➢ MEASURING AN INPUT SIZE :
- ♦ An algorithm's efficiency as a function of some parameter n indicating the algorithm's input size.
- ♦ In most cases, selecting such a parameter is quite straightforward.
- ♦ For example, it will be the size of the list for problems of sorting, searching, finding the list's smallest element, and most other problems dealing with lists.
- ♦ For the problem of evaluating a polynomial $p(x) = a_n x^n + \ldots + a_0$ of degree n, it will be the polynomial's degree or the number of its coefficients, which is larger by one than its degree.
- ♦ There are situations, of course, where the choice of a parameter indicating an input size does matter.
  - ♦ **Example** - computing the product of two n-by-n matrices.
  - ♦ There are two natural measures of size for this problem.
    - ♦ The matrix order n.
    - ♦ The total number of elements N in the matrices being multiplied.
- ♦ Since there is a simple formula relating these two measures, we can easily switch from one to the other, but the answer about an algorithm's efficiency will be qualitatively different depending on which of the two measures we use.
- ♦ The choice of an appropriate size metric can be influenced by operations of the algorithm in question. For example, how should we measure an input's size for a spell- checking algorithm? If the algorithm examines individual characters of its input, then we should measure the size by the number of characters; if it works by processing words, we should count their number in the input.
- ♦ We should make a special note about measuring size of inputs for algorithms involving properties of numbers (e.g., checking whether a given integer n is prime).
- ♦ For such algorithms, computer scientists prefer measuring size by the number b of bits in the n's binary representation:

$$b = \lfloor \log_2 n \rfloor + 1$$

- ♦ This metric usually gives a better idea about efficiency of algorithms in question.

➢ UNITS FOR MEASURING RUN TIME:
- ♦ We can simply use some standard unit of time measurement-a second, a millisecond, and so on-to measure the running time of a program implementing the algorithm.
- ♦ There are obvious drawbacks to such an approach. They are Dependence on the speed of a particular computer.Dependence on the quality of a program implementing the algorithm
- ♦ The compiler used in generating the machine code.The difficulty of clocking the actual running time of the program.
- ♦ Since we are in need to measure algorithm efficiency, we should have a metric that does not depend on these extraneous factors.
- ♦ One possible approach is to count the number of times each of the algorithm's operations is executed. This approach is both difficult and unnecessary.
- ♦ The main objective is to identify the most important operation of the algorithm, called the basic operation, the operation contributing the most to the total running time, and compute the number of times the basic operation is executed.
  - ➢ WORST CASE, BEST CASE AND AVERAGE CASE EFFICIENCES
- ♦ It is reasonable to measure an algorithm's efficiency as a function of a parameter indicating the size of the algorithm's input.

- But there are many algorithms for which running time depends not only on an input size but also on the specifics of a particular input.
- **Example, sequential search**. This is a straightforward algorithm that searches for a given item (some search key K) in a list of n elements by checking successive elements of the list until either a match with the search key is found or the list is exhausted.

- Here is the algorithm's pseudo code, in which, for simplicity, a list is implemented as an array. (It also assumes that the second condition A[i] i= K will not be checked if the first one, which checks that the array's index does not exceed its upper bound, fails.)

ALGORITHM Sequential Search(A[0..n -1], K)

//Searches for a given value in a given array by sequential search

//Input: An array A[0..n -1] and a search key K

//Output: Returns the index of the first element of A that matches K

//   or -1 if there are no matching elements i←0

 while i < n and A[i] ≠ K **do**

      i←i+1

if i < n **return** i

else return **-1**

> Worst case efficiency

- The worst-case efficiency of an algorithm is its efficiency for the worst-case input of size n, which is an input (or inputs) of size n for which the algorithm runs the longest among all possible inputs of that size.
- In the worst case, when there are no matching elements or the first matching element happens to be the last one on the list, the algorithm makes the largest number of key comparisons among all possible inputs of size n:

$$C_{worst}(n) = n.$$

- To analyze the algorithm to see what kind of inputs yield the largest value of the basic operation's count C(n) among all possible inputs of size n and then compute this worst- case value $C_{worst}(n)$
- The worst-case analysis provides very important information about an algorithm's efficiency by bounding its running time from above. In other words, it guarantees that for any instance of size n, the running time will not exceed $C_{worst}(n)$ its running time on the worst-case inputs.

> Best case Efficiency

- The best-case efficiency of an algorithm is its efficiency for the best-case input of size n, which is an input (or inputs) of size n for which the algorithm runs the fastest among all possible inputs of that size.
- We can analyze the best case efficiency as follows.
- First, determine the kind of inputs for which the count C (n) will be the smallest among all possible inputs of size n. (Note that the best case does not mean the smallest input; it means the input of size n for which the algorithm runs the fastest.)
- Then ascertain the value of C (n) on these most convenient inputs.
- Example- for sequential search, best-case inputs will be lists of size n with their first elements equal to a search key; accordingly, $C_{best}(n) = 1$.

> Average case efficiency

- It yields the information about an algorithm about an algorithm's behaviour on a

‒typical‖ and ‒random‖ input.

♦ To analyze the algorithm's average-case efficiency, we must make some assumptions about possible inputs of size n.

The average number of key comparisions $C_{avg}(n)$ can be computed as follows,

♦ let us consider again sequential search. The standard assumptions are,

♦ In the case of a successful search, the probability of the first match occurring in the ith position of the list is pin for every i, and the number of comparisons made by the algorithm in such a situation is obviously i.

♦ In the case of an unsuccessful search, the number of comparisons is n with the probability of such a search being (1 - p). Therefore,

$$C_{avg}(n) = \left[1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \dots + i \cdot \frac{p}{n} + \dots + n \cdot \frac{p}{n} \right] + n \cdot (1 - p)$$

$$= \frac{p}{n} [1 + 2 + 3 + \dots + i + \dots + n] + n (1 - p)$$

$$= \frac{p}{n} \frac{n(n+1)}{2} + n(1 - p)$$

$$= \frac{p(n+1)}{2} + n(1 - p)$$

♦ Example, if p = 1 (i.e., the search must be successful), the average number of key comparisons made by sequential search is (n + 1) /2.

♦ If p = 0 (i.e., the search must be unsuccessful), the average number of key comparisons will be n because the algorithm will inspect all n elements on all such inputs.

Recapitulation of the Analysis Framework

➢ Both time and space efficiencies are measured as functions of the algorithm's input size.

➢ Time efficiency is measured by counting the number of times the algorithm's basic operation is executed. Space efficiency is measured by counting the number of extra memory units consumed by the algorithm.

➢ The efficiencies of some algorithms may differ significantly for inputs of the same size.

➢ The framework's primary interest lies in the order of growth of the algorithm's running time (extra memory units consumed) as its input size goes to infinity.

Recurrence relation: substitution method

A recurrence is an equation or inequality that describes a function in terms of its values on smaller inputs. To solve a Recurrence Relation means to obtain a function defined on the natural numbers that satisfy the recurrence.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1, \end{cases}$$

There are four methods for solving Recurrence:
1) Substitution Method.
   a) Backward Substitution. eg) X(n) = X(n-1) + 5
   b) Forward Substitution eg) X(n) = X(n+1) + 5
2) Recursion Tree Method.
3) Master Theorem Method.

1. Substitution Method:

The Substitution Method Consists of two main steps:

1. Guess the Solution.
2. Use the mathematical induction to find the boundary condition and shows that the guess is correct.

Forward substitution:

One of the simplest methods for solving simple recurrence relations is using forward substitution. In this method, we solve the recurrence relation for n=0,1,2,…until we see a pattern. Then we make a guesswork and predict the running time. The final and important step in this method is we need to verify that our guesswork is correct by using the induction.

Example:

$T(n) = 1$ **if** n=1

$= 2T(n-1) + 1$ **if** n>1

Solution:

$T(1) = 1$

$T(2) = 2\ T(2-1) + 1 = 2T(1) + 1 = 2(1) + 1 = 3$

$T(3) = 2\ T(3-1) + 1 = 2T(2) + 1 = 2(3) + 1 = 7$

$T(4) = 2\ T(4-1) + 1 = 2T(3) + 1 = 2(7) + 1 = 15$

⇨ **$T(n) = 2^n - 1$**

Backward substitution:

In backward substitution, we do the opposite i.e. we put n=n,n−1,n−2,.. or n=n,n/2,n/4,… until we see the pattern. After we see the pattern, we make a guesswork for the running time and we verify the guesswork.

Example:

$X(n) = X(n-1) + 5$ for n>1 X(1)=0

Solution:

$X(n) = X(n-1) + 5$ -(1)

$X(n-1) = X(n-2) + 5$ -(2)

Substitute (2) in (1)

$X(n) = X(n-2) + 5 + 5$ -(3)

$X(n-2) = X(n-3) + 5$ -(4)

Substitute (4) in (3)

$X(n) = X(n-3) + 3 \times 5$

⇨ $X(n) = X(n-i) + i \times 5$

n-i = 1 as per initial condition

i = n-1

⇨ $X(n) = X(n-(n-1)) + (n-1) \times 5$

$$=X(1)+5(n-1)$$

$$X(n)=5(n-1) \quad \text{where } X(1)=0$$

2.Recursive Tree Method:

1. Recursion Tree Method is a pictorial representation of an iteration method which is in the form of a tree where at each level nodes are expanded.
2. In general, we consider the second term in recurrence as root.
3. It is useful when the divide & Conquer algorithm is used.
4. It is sometimes difficult to come up with a good guess. In Recursion tree, each root and child represents the cost of a single subproblem.
5. We sum the costs within each of the levels of the tree to obtain a set of pre-level costs and then sum all pre-level costs to determine the total cost of all levels of the recursion.
6. A Recursion Tree is best used to generate a good guess, which can be verified by the Substitution Method.

$$\text{Consider } T(n) = 2T\left(\frac{n}{2}\right) + n^2$$

Solution: The Recursion tree for the above recurrence is
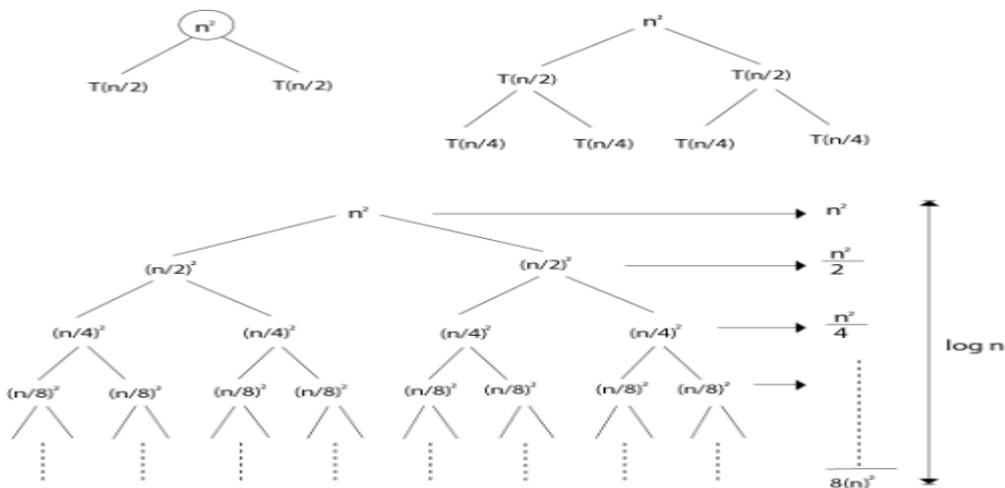
$$T(n) = n^2 + \frac{n^2}{2} + \frac{n^2}{4} + \ldots \ldots \log n \text{ times.}$$

$$\leq n^2 \sum_{i=0}^{\infty} \left(\frac{1}{2^i}\right)$$

$$\leq n^2 \left(\frac{1}{1-\frac{1}{2}}\right) \leq 2n^2$$

$$T(n) = \theta n^2$$

Master Theorem:

Master Theorem If $f(n) \in \Theta(n^d)$ where $d \geq 0$ in recurrence (5.1), then

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d, \\ \Theta(n^d \log n) & \text{if } a = b^d, \\ \Theta(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

Analogous results hold for the O and Ω notations, too.

For example, the recurrence for the number of additions $A(n)$ made by the divide-and-conquer sum-computation algorithm (see above) on inputs of size $n = 2^k$ is

$$A(n) = 2A(n/2) + 1.$$

Thus, for this example, $a = 2$, $b = 2$, and $d = 0$; hence, since $a > b^d$,

$$A(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n).$$

If, $C(n) = 2C(n/2) + C_{merge}(n)$ for $n > 1$, $C(1) = 0$.

for the worst case, $C_{merge}(n) = n - 1$, and we have the recurrence

$$C_{worst}(n) = 2C_{worst}(n/2) + n - 1 \quad \text{for } n > 1, \quad C_{worst}(1) = 0$$

a=2,b=2,d=0, then, C(n)= **θ(n log n)**

Lower bounds:

Lower bound in algorithms refers to the minimum amount of resources (such as time or space) that an algorithm must take to solve a particular problem. Lower bounds are used to establish a lower limit on the performance of any algorithm for a specific problem, which helps to determine if a given algorithm is optimal or if there is room for improvement.

There are several ways to establish lower bounds, including:

1. Information-theoretic lower bounds: These lower bounds are based on the information-theoretic limits of computation, and are independent of the specific algorithm being used. For example, the lower bound for sorting n items is Ω(n log n).
2. Computational lower bounds: These lower bounds are based on the assumption that certain computational problems are hard, and that any algorithm for solving those problems must take a certain amount of time.

3. Adversarial lower bounds: These lower bounds are based on the worst-case scenario, where an adversary is trying to make the algorithm perform as poorly as possible. For example, the lower bound for searching an unsorted list of n items is Ω(n).

The lower bound provides a baseline for the performance of any algorithm, and helps to determine the best possible solution for a given problem. By knowing the lower bound, researchers and practitioners can evaluate different algorithms and determine which one is the most efficient for a specific problem.

Searching :

Searching is an operation or a technique that helps finds the place of a given element or value in the list. Any search is said to be successful or unsuccessful depending upon whether the element that is being searched is found or not. Some of the standard searching technique that is being followed in data structure is listed below:

1. Linear Search

2. Binary Search

LINEAR SEARCH

Linear search is a very basic and simple search algorithm. In Linear search, we search an element or value in a given array by traversing the array from the starting, till the desired element or value is found.

It compares the element to be searched with all the elements present in the array and when the element is **matched** successfully, it returns the index of the element in the array, else it return -1.

Linear Search is applied on unsorted or unordered lists, when there are fewer elements in a list.For

Example,
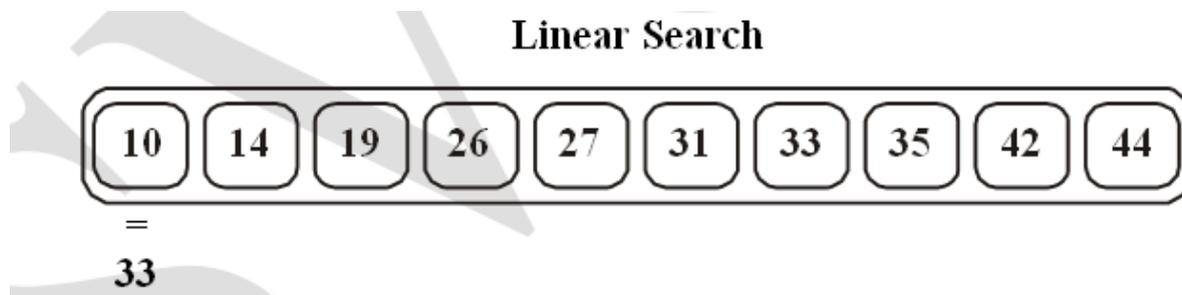
Linear Search

| 10 | 14 | 19 | 26 | 27 | 31 | 33 | 35 | 42 | 44 |

=

33

ALGORITHM *SequentialSearch*(A[0..n − 1], K)
//Searches for a given value in a given array by sequential search
//Input: An array A[0..n − 1] and a search key K
//Output: The index of the first element in A that matches K
//   or −1 if there are no matching elements
i ← 0
while i < n **and** A[i] ≠ K **do**
    i ← i + 1
if i < n **return** i
else return −1

Advantages of Linear Searching:
- It is simple to implement.
- It does not require specific ordering before applying the method

Disadvantages of Linear searching:
- It is less efficient

Time Complexity:

The best-case inputs for sequential search are lists of size n with their first element equal to a search key; accordingly, Cbest(n) = 1

In the case of an unsuccessful search, the number of comparisons will be n with the probability of such a search being (1 − p). Therefore

$$C_{avg}(n) = [1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \cdots + i \cdot \frac{p}{n} + \cdots + n \cdot \frac{p}{n}] + n \cdot (1 - p)$$

$$= \frac{p}{n}[1 + 2 + \cdots + i + \cdots + n] + n(1 - p)$$

$$= \frac{p}{n} \frac{n(n + 1)}{2} + n(1 - p) = \frac{p(n + 1)}{2} + n(1 - p).$$

Features of Linear Search Algorithm

1. It is used for unsorted and unordered small list of elements.

2. It has a time complexity of **O(n)**, which means the time is linearly dependent on the number of elements, which is not bad, but not that good too.

3. It has a very simple implementation.

BINARY SEARCH

Binary Search is used with sorted array or list. In binary search, we follow the following steps:

1. We start by comparing the element to be searched with the element in the middle of the list/array.

2. If we get a match, we return the index of the middle element.

3. If we do not get a match, we check whether the element to be searched is less or greater than in value than the middle element.

4. If the element/number to be searched is greater in value than the middle number, then we pick the elements on the right side of the middle element(as the list/array is sorted, hence on the right, we will have all the numbers greater than the middle number), and start again from the step 1.

5. If the element/number to be searched is lesser in value than the middle number, then we pick the elements on the left side of the middle element, and start again from the step 1.

Features of Binary Search

1. It is great to search through large sorted arrays.

2. It has a time complexity of **O(log n)** which is a very good time complexity. It has a simple implementation.

Binary search is a fast search algorithm with run-time complexity of Ï(log n). This search algorithm works on the principle of divide and conquers. For this algorithm to work properly, the data collection should be in the sorted form.

Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched

for in the sub-array to the right of the middle item. This process continues on the sub- array as well until the size of the sub array reduces to zero.

How Binary Search Works?

For a binary search to work, it is mandatory for the target array to be sorted. We shall learnthe process of binary search with a pictorial example. The following is our sorted array and let usassume that we need to search the location of value 31 using binary search.

10	14	19	26	27	31	33	35	42	44
0	1	2	3	4	5	6	7	8	9

First, we shall determine half of the array by using this formula -mid

$= low + (high - low) / 2$

Here it is, $0 + (9 - 0) / 2 = 4$ (integer value of 4.5). So, 4 is the mid of the array.

10	14	19	26	27	31	33	35	42	44
0	1	2	3	4	5	6	7	8	9

Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.

10	14	19	26	27	31	33	35	42	44
0	1	2	3	4	5	6	7	8	9

We change our low to mid + 1 and find the new mid value again.low

$= mid + 1$

$mid = low + (high - low) / 2$

Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.

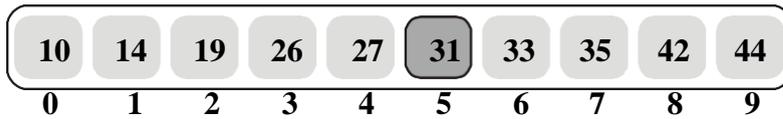| 10	14	19	26	27	31	33	35	42	44
0	1	2	3	4	5	6	7	8	9

The value stored at location 7 is not a match, rather it is more than what we are lookingfor. So, the value must be in the lower part from this location.

10	14	19	26	27	31	33	35	42	44
0	1	2	3	4	5	6	7	8	9

Hence, we calculate the mid again. This time it is 5.

10	14	19	26	27	31	33	35	42	44
0	1	2	3	4	5	6	7	8	9

We compare the value stored at location 5 with our target value. We find that it is a match.

10	14	19	26	27	31	33	35	42	44
0	1	2	3	4	5	6	7	8	9

We change our low to mid + 1 and find the new mid value again.low

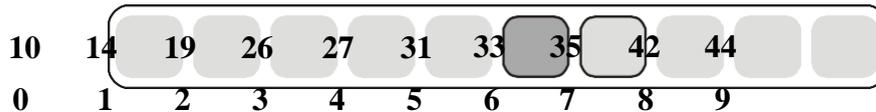= mid + 1

mid = low + (high - low) / 2

Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.

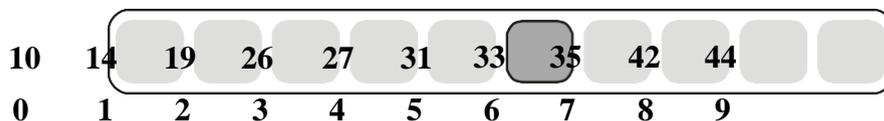| 10	14	19	26	27	31	33	35	42	44
0	1	2	3	4	5	6	7	8	9

The value stored at location 7 is not a match, rather it is more than what we are lookingfor. So, the value must be in the lower part from this location.

10	14	19	26	27	31	33	35	42	44
0	1	2	3	4	5	6	7	8	9

Hence, we calculate the mid again. This time it is 5.

10	14	19	26	27	31	33	35	42	44
0	1	2	3	4	5	6	7	8	9

We compare the value stored at location 5 with our target value. We find that it is a match.

10	14	19	26	27	31	33	35	42	44
0	1	2	3	4	5	6	7	8	9

We conclude that the target value 31 is stored at location 5.

ALGORITHM *BinarySearch*$(A[0..n-1], K)$

    //Implements nonrecursive binary search
    //Input: An array $A[0..n-1]$ sorted in ascending order and
    //   a search key K
    //Output: An index of the array's element that is equal to K
    //   or -1 if there is no such element
    $l \leftarrow 0; \quad r \leftarrow n-1$
    while $l \leq r$ **do**
        $m \leftarrow \lfloor (l+r)/2 \rfloor$
        if $K = A[m]$ **return** m
        else if $K < A[m] \ r \leftarrow m-1$
        else $l \leftarrow m+1$
    return -1

How many such comparisons does the algorithm make on an array of n elements? The answer obviously depends not only on n but also on the specifics of a particular instance of the problem. Let us find the number of key comparisons in the worst case $C_{worst}(n)$. The worst-case inputs include all arrays that do not contain a given search key, as well as some successful searches. Since after one comparison the algorithm faces the same situation but for an array half the size, we get the following recurrence relation for $C_{worst}(n)$:

$$C_{worst}(n) = C_{worst}(\lfloor n/2 \rfloor) + 1 \quad \text{for } n > 1, \quad C_{worst}(1) = 1.$$

$$C_{worst}(2^k) = k+1 = \log_2 n + 1.$$

$$C_{worst}(n) = \lfloor \log_2 n \rfloor + 1 = \lceil \log_2(n+1) \rceil.$$

$$C_{avg}(n) \approx \log_2 n.$$

Interpolation Search:

Interpolation search is an improved variant of binary search. This search algorithm works on the probing position of required value. For this algorithm to work properly the data collection should be in sorted form and equally distributed.

Binary search has huge advantage of time complexity over linear search. Linear search has worstcase complexity of $O(n)$ whereas binary search has $O(\log n)$. There are cases where the location of target data may be known in advance.

For example, in case of telephone directory, if we want to search telephone number of Morphius. Here, linear search and even binary search will seem slow as we can directly jump to memory space where names start from 'M' are stored.

Position Probing in Interpolation Search

Interpolation search search a particular item by computing the probe position. Initially probe position is the position of the middle most item of the collection.

1	2	3	4	5	6	7	8	9	10

1	2	3	4	5	6	7	8	9	10

If match occurs then index of item is returned. To split the list into two parts we use the following method

```
mid = Lo + ((Hi - Lo) / (A[Hi] - A[Lo])) * (X - A[Lo])

where -
    A    = list
    Lo   = Lowest index of the list
    Hi   = Highest index of the list
    A[n] = Value stored at index n in the list
```

If middle item is greater than item then probe position is again calculated in the sub-array to the right of the middle item other wise item is search in sub-array to the left of the middle item. This process continues on sub-array as well until the size of subarray reduces to zero. Runtime complexity of interpolation search algorithm is Olog(logn) as compared to Ologn of BST in favourable situations

Algorithm:

```
Step 1  - Start searching data from middle of the list.
Step 2  - If it is a match, return the index of the item, and exit.
Step 3  - If it is not a match, probe position.
Step 4  - Divide the list using probing formula and find the new midle.
Step 5  - If data is greater than middle, search in higher sub-list.
Step 6  - If data is smaller than middle, search in lower sub-list.
Step 7  - Repeat until match.
```

Pseudocode:

```
A → Array list
N → Size of A
X → Target Value

Procedure Interpolation_Search()

    Set Lo  →   0
    Set Mid → -1
    Set Hi  →   N-1

    While X does not match

        if Lo equals to Hi OR A[Lo] equals to A[Hi]
           EXIT: Failure, Target not found
        end if

        Set Mid = Lo + ((Hi - Lo) / (A[Hi] - A[Lo])) * (X - A[Lo])

        if A[Mid] = X
           EXIT: Success, Target found at Mid
        else
           if A[Mid] < X
              Set Lo to Mid+1
           else if A[Mid] > X
              Set Hi to Mid-1
           end if
        end if

    End While

End Procedure
```

Pattern search/ Pattern Searching or String matching

String matching algorithms are normally used in text processing. Normally text processing is done in compilation of program.

In software design or system design also text processing is a vital activity.

Given a string of n characters called the text and a string of m characters (m ≤ n) called the pattern, find a substring of the text that matches the pattern.

To put it more precisely, we want to find i—the index of the leftmost character of the first matching substring in the text—such that ti = p0,...,ti+j = pj ,...,ti+m−1 = pm−1:

$$t_0 \quad \cdots \quad t_i \quad \cdots \quad t_{i+j} \quad \cdots \quad t_{i+m-1} \quad \cdots \quad t_{n-1} \qquad \text{text } T$$
$$\updownarrow \qquad\qquad \updownarrow \qquad\qquad \updownarrow$$
$$p_0 \quad \cdots \quad p_j \quad \cdots \quad p_{m-1} \qquad \text{pattern } P$$

If matches other than the first one need to be found, a string-matching algorithm can simply continue working until the entire text is exhausted.

Different string matching Algorithms are,
1. The naïve string- matching algorithm
2. Rabin-Karp algorithm
3. Knuth-Morris-Pratt algorithm

The naïve string- matching algorithm:

The naive algorithm finds all valid shifts using a loop that checks the condition $P[1..m] = T[s+1..s+m]$ for each of the $n - m + 1$ possible values of s.

NAIVE-STRING-MATCHER (T, P)

```
1   n = T.length
2   m = P.length
3   for s = 0 to n − m
4       if P[1..m] == T[s + 1..s + m]
5           print "Pattern occurs with shift" s
```

The naive string-matching procedure as sliding a "template" containing the pattern over the text, noting for which shifts all of the characters on the template equal the corresponding characters in the text.

The for loop of lines 3–5 considers each possible shift explicitly. The test in line 4 determines whether the current shift is valid; this test implicitly loops to check corresponding character positions until all positions match successfully or a mismatch is found. Line 5 prints out each valid shift s.

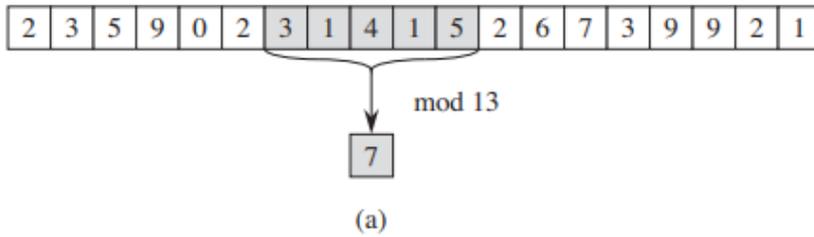Procedure NAIVE-STRING-MATCHER takes time O((n - m +1)m), and this bound is tight in the worst case. For example, consider the text string an (a string of n a's) and the pattern am. For each of the nmC1 possible values of the shift s, the implicit loop on line 4 to compare corresponding characters must execute m times to validate the shift.

(a)   (b)   (c)   (d)
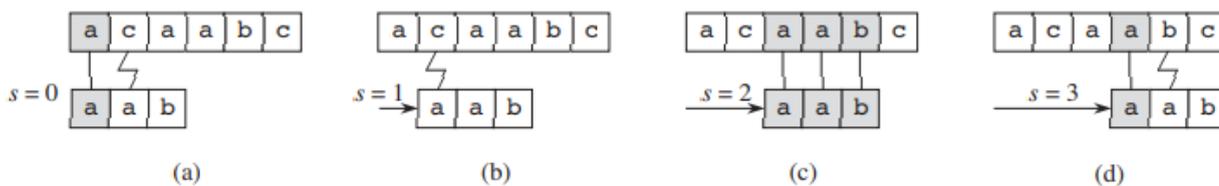
The Rabin-Karp algorithm:

The Rabin – Karp method is based on hashing technique. The algorithm assumes each character to be a digit in radix-d notation. It makes use of equivalence of two numbers modulo a third number.

Given a pattern P[1..m], let p denote its corresponding decimal value. In a similar manner, given a text T[1..n], let ts denote the decimal value of the length-m substring T[s+1..s+m], for s =0,1….n - m. Certainly, ts=p if and only if T[s+1..s+m]=P[1..m] thus, s is a valid shift if and only if ts=p.

2 3 5 9 0 2 3 1 4 1 5 2 6 7 3 9 9 2 1

mod 13

7

(a)

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |

2 3 5 9 0 2 3 1 4 1 5 2 6 7 3 9 9 2 1

... ... mod 13

8 9 3 11 0 1 7 8 4 5 10 11 7 9 11

valid match

spurious hit

(b)

old high-order digit   new low-order digit   old high-order digit   shift   new low-order digit

3 1 4 1 5 2

$$14152 \equiv (31415 - 3 \cdot 10000) \cdot 10 + 2 \pmod{13}$$
$$\equiv (7 - 3 \cdot 3) \cdot 10 + 2 \pmod{13}$$
$$\equiv 8 \pmod{13}$$

7 8

(c)
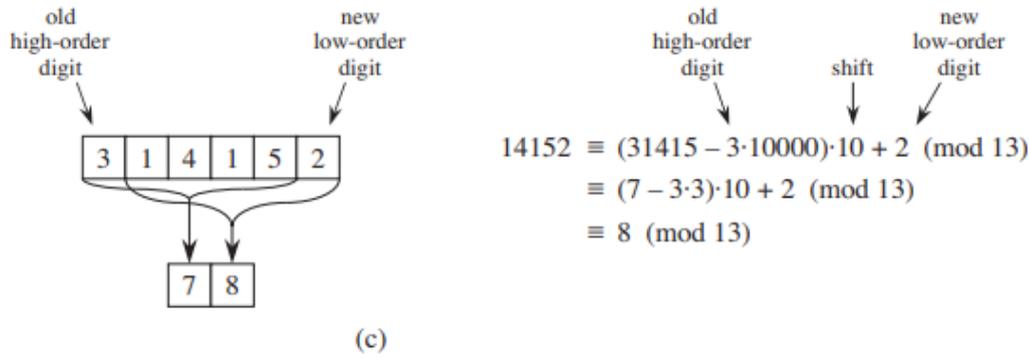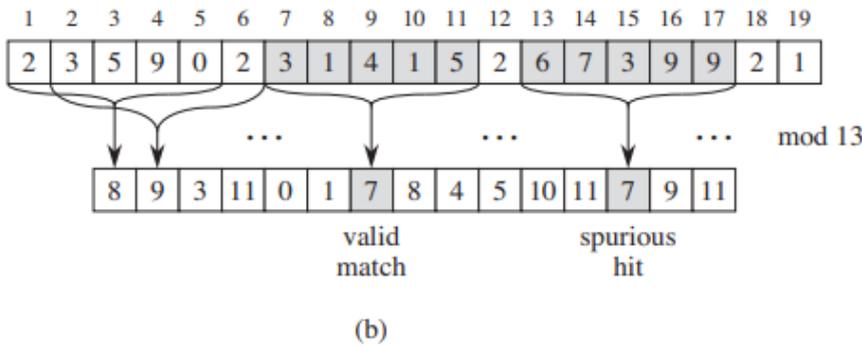
Algorithm:

RABIN-KARP-MATCHER(T, P, d, q)

```
1   n = T.length
2   m = P.length
3   h = d^(m-1) mod q
4   p = 0
5   t_0 = 0
6   for i = 1 to m                      // preprocessing
7       p = (dp + P[i]) mod q
8       t_0 = (dt_0 + T[i]) mod q
9   for s = 0 to n − m                   // matching
10      if p == t_s
11          if P[1..m] == T[s + 1..s + m]
12              print "Pattern occurs with shift" s
13      if s < n − m
14          t_{s+1} = (d(t_s − T[s + 1]h) + T[s + m + 1]) mod q
```

The Knuth-Morris-Pratt algorithm:

- In pattern matching algorithms, we often compare the pattern characters that do not match in the text and on occurrence of mismatch we simply through away the information and restart the comparison, for another set of characters from the text.
- Thus again and again with next incremental position of the text, the characters from pattern are matched. This ultimately reduces the efficiency of the pattern matching algorithm. Hence Knuth- Morris- Pratt algorithm came up which avoids the repeated comparison of characters.
- The algorithm is named after the scientists Knuth, Morris and Pratt.
- The basic idea behind the algorithm is to build **Prefix table.** It is also called array or failure function table.
- This prefix table is build using prefix and suffix information of pattern.
- The overlapping prefix and suffix is used in KMP algorithm.

Given that pattern characters $P[1..q]$ match text characters $T[s+1..s+q]$, what is the least shift $s' > s$ such that for some $k < q$,

Given text: abcxabcdabxabcdabcdabcy

Given pattern : abcdabcy

Step 1: we will construct the prefix table for the given pattern as follows.

0	1	2	3	4	5	6	7
a	b	c	D	a	b	c	Y
0	0	0	0	1	2	3	0

Step2: now start matching search for pattern against the text with the help of prefix table.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
a	b	c	x	a	b	c	d	a	b	x	a	b	c	d	a	b	c	d	a	B	c	Y
√	√	√	x																			
a	b	c	d	a	b	c	y															

The pattern[3] is not matching with text[3]. Hence we find the position using the formula,

Text index of unmatched character – prefixtable[pattern index – 1]

=3 – prefixtable[3-1]

=3- 0

= 3

That means shift pattern at starting index 3.

Step 3:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
a	b	c	x	a	b	c	d	a	b	X	a	b	c	d	a	b	c	d	a	B	c	Y
			x																			
			a	b	c	d	a	b	c	Y												

As pattern[0] is not matching with text[3], so shift pattern by one position.

Step 4:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
a	b	c	x	a	b	c	d	a	b	X	a	b	c	d	a	b	c	d	a	B	c	Y
				√	√	√	√	√	√	x												
				a	b	c	d	a	b	c	y											

Text index of unmatched character – prefixtable[pattern index – 1]

=10 – prefixtable[6-1]

=10- 2

= 8

Step 5:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
a	b	c	x	a	b	c	d	a	b	X	a	b	c	d	a	b	c	d	a	B	c	Y
								√	√	x												
								a	b	c	d	a	b	c	y							

Text index of unmatched character – prefixtable[pattern index – 1]

=10 – prefixtable[2-1]

=10- 0

= 10

Step 6:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
a	b	c	x	a	b	c	d	a	b	X	a	b	c	d	a	b	c	d	a	B	c	Y
										x												
										a	b	c	d	a	b	c	y					

Text[10] is not matching with pattern[0]. Hence shift one position next.

Step 7:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
a	b	c	x	a	b	c	d	a	b	X	a	b	c	d	a	b	c	d	a	B	c	Y
											√	√	√	√	√	√	√	x				

									a	b	c	d	a	b	c	y				

Text index of unmatched character – prefixtable[pattern index – 1]

=18 – prefixtable[7-1]

=18- 3

= 15

Step 8:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
a	b	c	x	a	b	c	d	a	b	X	a	b	c	d	a	b	c	d	a	B	c	Y
															√	√	√	√	√	√	√	√
															a	b	c	d	a	B	c	Y

Thus we found the pattern at starting position 15 in text string.

Insertion sorting:

Insertion sort is usually done by scanning the sorted subarray from right to left until the first element smaller than or equal to A[n − 1] is encountered to insert A[n − 1] right after that element. Starting with A[1] and ending with A[n − 1], A[i]is inserted in its appropriate place among the first i elements of the array that have been already sorted.

```
ALGORITHM   InsertionSort(A[0..n − 1])
    //Sorts a given array by insertion sort
    //Input: An array A[0..n − 1] of n orderable elements
    //Output: Array A[0..n − 1] sorted in nondecreasing order
    for i ← 1 to n − 1 do
        v ← A[i]
        j ← i − 1
        while j ≥ 0 and A[j] > v do
            A[j + 1] ← A[j]
            j ← j − 1
        A[j + 1] ← v
```

$$A[0] \le \cdots \le A[j] < A[j + 1] \le \cdots \le A[i − 1] \mid A[i] \cdots A[n − 1]$$

smaller than or equal to $A[i]$ greater than $A[i]$
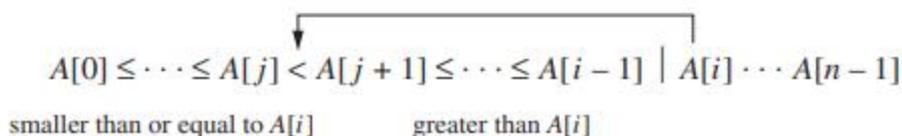
```
89 | 45   68   90   29   34   17
45   89 | 68   90   29   34   17
45   68   89 | 90   29   34   17
45   68   89   90 | 29   34   17
29   45   68   89   90 | 34   17
29   34   45   68   89   90 | 17
17   29   34   45   68   89   90
```

Example of sorting with insertion sort. A vertical bar separates the sorted part of the array from the remaining elements; the element being inserted is in bold.

The number of key comparisons in this algorithm obviously depends on the nature of the input. In the worst case, $A[j] > v$ is executed the largest number of times, i.e., for every $j = i - 1, \ldots, 0$. Since $v = A[i]$, it happens if and only if $A[j] > A[i]$ for $j = i - 1, \ldots, 0$. (Note that we are using the fact that on the ith iteration of insertion sort all the elements preceding $A[i]$ are the first i elements in the input, albeit in the sorted order.) Thus, for the worst-case input, we get $A[0] > A[1]$ (for $i = 1$), $A[1] > A[2]$ (for $i = 2$), $\ldots$, $A[n - 2] > A[n - 1]$ (for $i = n - 1$). In other words, the worst-case input is an array of strictly decreasing values. The number of key comparisons for such an input is

$$C_{worst}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \in \Theta(n^2).$$

Heap sort:

Heapsort is a comparison-based sorting algorithm that works by building a binary heap data structure and extracting the maximum element from the heap.

Here is a step-by-step algorithm for the heapsort process:

1. Build a binary heap from the input data:
    *   Start by treating the input data as a complete binary tree.
    *   Compare the value of each node with its children and swap the node with the larger child, if necessary, to maintain the "max heap" property.
    *   Repeat the above step for each node in the tree until the root node holds the largest value.
2. Extract the maximum element from the heap:
    *   Swap the root node with the last element in the heap.
    *   Remove the last element from the heap, as it is now in its correct position in the sorted data.
    *   Compare the value of the root node with its children and swap the node with the larger child, if necessary, to maintain the "max heap" property.
    *   Repeat the above step until the heap is a complete binary tree with all nodes in the correct order.
3. Repeat the above steps until all elements have been extracted and placed in their correct position in the sorted data.

The running time of heapsort is O(n log n), making it a fast and efficient sorting algorithm for large datasets.

UNIT II   GRAPH ALGORITHMS

Graph algorithms: Representations of graphs - Graph traversal: DFS – BFS - applications - Connectivity,strong connectivity, bi-connectivity - Minimum spanning tree: Kruskal's and Prim's algorithm- Shortest path: Bellman-Ford algorithm - Dijkstra's algorithm - Floyd-Warshall algorithm Network flow: Flow networks - Ford-Fulkerson method – Matching: Maximum bipartite matching

Representations of graphs :

A graph can be represented in several ways, including:

1.   Adjacency matrix / Incidence Matrix

2.   Adjacency Linked List/ Incidence Linked List

Adjacency Matrix: It is a two-dimensional matrix of size V x V, where V is the number of vertices in the graph, and each cell $a_{i,j}$ represents the weight of an edge between vertex iand vertex j. If there is no edge between vertex i and vertex j, then the value of $a_{i,j}$ is set to infinity.

For directed graph

A[u][v]= {   1, if there is edge from u to v
        0   otherwise   }

For undirected graph

A[u][v]= {   1, if there is edge between u and v
        0   otherwise   }

For weighted graph

A[u][v]= {   value , if there is edge from u to v
∞, if no edge between u and v   }

Adjacency List: In this representation, each vertex is stored as an object that contains a list of its neighbors along with the weight of the edges connecting it to them. This representation is useful when the graph is sparse, i.e., it has a small number of edges compared to the number of vertices.
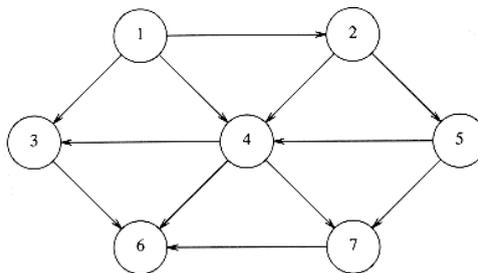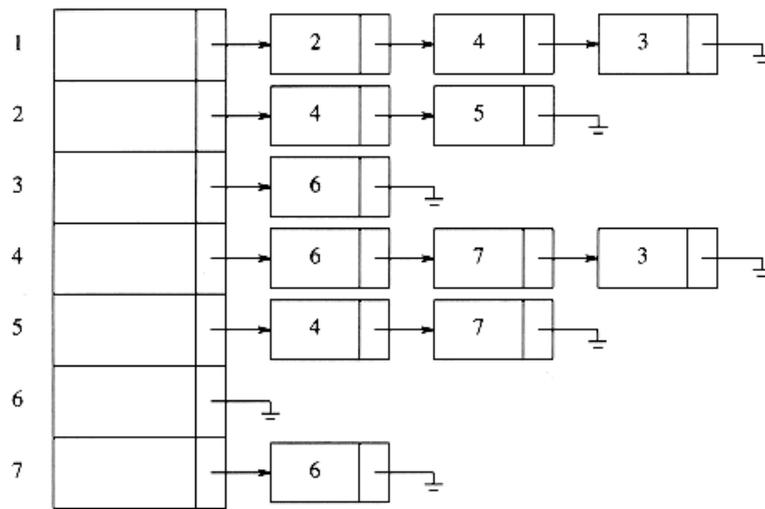
Graph Traversal:

Visiting of each and every vertex in the graph only once is called as **Graph traversal.**

There are two types of Graph traversal.

1.   Depth First Traversal/ Search (DFS)
2.   Breadth First Traversal/ Search (BFS)

Depth First Traversal/ Search
(DFS)

Depth-first search is a generalization of preorder traversal. Starting at some vertex, v, we process v and then recursively traverse all vertices adjacent to v. If this process is performed on a tree, then all tree vertices are systematically visited in a total of O(|E|) time, since |E| = (|V|).

We need to be careful to avoid cycles. To do this, when we visit a vertex v, we mark it visited, since now we have been there, and recursively call depth-first search on all adjacent vertices that are not already marked.

The two important key points of depth first search

1. If path exists from one node to another node walk across the edge – **exploring the edge**

2. If path does not exist from one specific node to any other nodes, return to the previous node where we have been before – **backtracking**

Procedure for DFS

Starting at some vertex V, we process V and then recursively traverse all the vertices adjacent to V. This process continues until all the vertices are processed. If some vertex is not processed recursively, then it will be processed by using backtracking. If vertex W is visited from V, then the vertices are connected by means of tree edges. If the edges not included in tree, then they are represented by back edges. At the end of this process, it will construct a tree called as DFS tree.

Routine to perform a depth-first search void

```
void dfs( vertex v )
{
visited[v] = TRUE;
for each w adjacent to v
if( !visited[w] )
dfs( w );
}
```

The (global) boolean array visited[ ] is initialized to FALSE. By recursively calling the procedures only on nodes that have not been visited, we guarantee that we do not loop indefinitely.

* An efficient way of implementing this is to begin the depth-first search at v1. If we need to restart the depth-first search, we examine the sequence vk, vk + 1, . . . for an unmarked vertex,where vk - 1 is the vertex where the last depth-first search was started.

An undirected graph
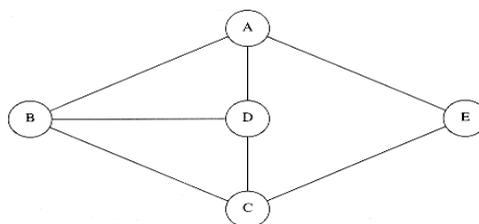
Steps to construct depth-first spanning tree

a.   We start at vertex A. Then we mark A as visited and call dfs(B) recursively. dfs(B) marks B as visited and calls dfs(C) recursively.

b.   dfs(C) marks C as visited and calls dfs(D) recursively.

c.   dfs(D) sees both A and B, but both these are marked, so no recursive calls are made. dfs(D) also sees that C is adjacent but marked, so no recursive call is made there, and dfs(D) returns back to dfs(C).

d.   dfs(C) sees B adjacent, ignores it, finds a previously unseen vertex E adjacent, and thus calls dfs(E).

e.   dfs(E) marks E, ignores A and C, and returns to dfs(C).

f.   dfs(C) returns to dfs(B). dfs(B) ignores both A and D and returns.

g.   dfs(A) ignores both D and E and returns.

---------> **Back edge**

⟶ **Tree edge**
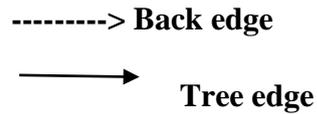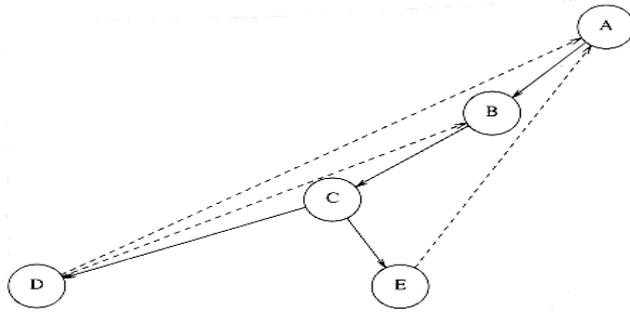
Depth Frist Search of the Graph

The root of the tree is A, the first vertex visited. Each edge (v, w) in the graph is present in the tree. If, when we process (v, w), we find that w is unmarked, or if, when we process (w, v), we find that v is unmarked, we indicate this with a **tree edge.**

If when we process (v, w), we find that w is already marked, and when processing (w, v), we find that v is already marked, we draw a dashed line, which we will call a **back edge**, to indicate that this "edge" is not really part of the tree.

Breadth First Traversal (BFS)

Here starting from some vertex v, and its adjacency vertices are processed. After all the adjacency vertices are processed, then selecting any one the adjacency vertex and process will continue. If the vertex is not visited, then backtracking is applied to visit the unvisited vertex.

Routine: Example: BFS of the above graph

void BFS (vertex v)

{

visited[v]= true;

For each w adjacent to v

If (!visited[w])

visited[w] = true;

}
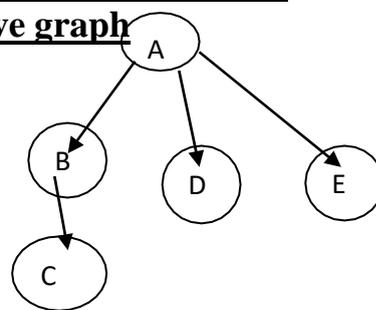
Difference between DFS & BFS

S. No	DFS	BFS
1	Back tracking is possible from a dead end.	Back tracking is not possible.
2	Vertices from which exploration is incomplete are processed in a LIFO order.	The vertices to be explored are organized as a FIFO queue.

3	Search is done in one particular direction at the time.	The vertices in the same level are maintained parallel. (Left to right) (alphabetical ordering)
4	 Order of traversal: A→B→C→D→E	 Order of traversal: A→B→C→D→E→F→G→H
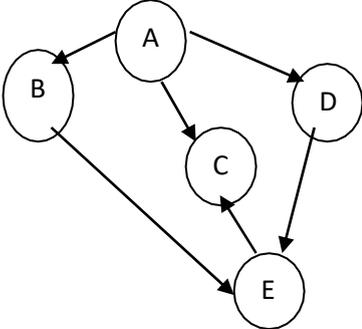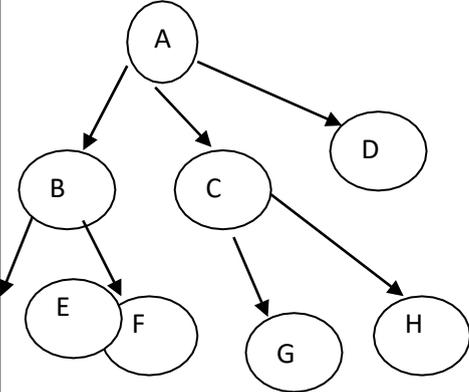
Minimum spanning tree

DEFINITION A spanning tree of an undirected connected graph is its connected acyclic subgraph (i.e., a tree) that contains all the vertices of the graph. If such a graph has weights assigned to its edges, a minimum spanning tree is its spanning tree of the smallest weight, where the weight of a tree is defined as the sum of the weights on all its edges. The minimum spanning tree problem is the problem of finding a minimum spanning tree for a given weighted connected graph
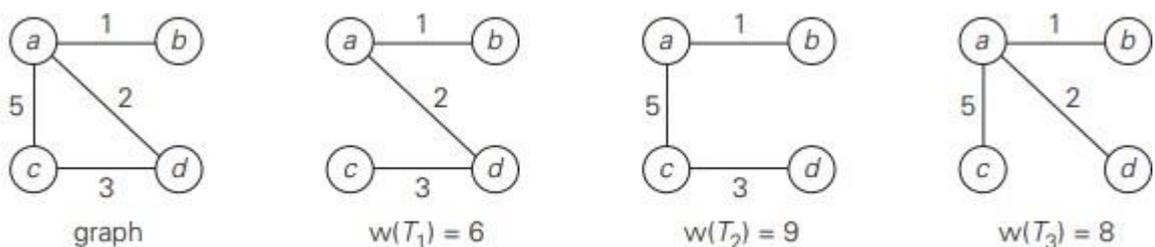
FIGURE 9.2 Graph and its spanning trees, with T_1 being the minimum spanning tree.

<u>Algorithms for Minimum Spanning Tree:</u>

1. Prim's algorithm

2. Kruskal's algorithm

Prim's algorithm

Prim's algorithm is a greedy algorithm used to find the minimum spanning tree of a graph. The algorithm starts with a random vertex and adds edges to the tree until all vertices are included in the tree. The edges are chosen in such a way that the sum of their weights is minimized.

Here's the steps to implement Prim's algorithm:

1. Create a set to keep track of all the vertices that have been added to the tree. Initially, this set only contains the starting vertex.

2. Create a priority queue to store the edges that connect the vertices in the tree to the vertices outside the tree. The priority of each edge is determined by its weight.

3. While the set of vertices in the tree is not equal to the total number of vertices in the graph, repeat the following steps:

    a. Remove the edge with the smallest weight from the priority queue.

    b. If both vertices of the edge are not in the set of vertices in the tree, add the edge to the tree and add both vertices to the set.

    c. For each vertex that was just added to the set, add all edges that connect that vertex to a vertex outside the set to the priority queue.

4. The final set of edges is the minimum spanning tree.

- Move $u*$ from the set $V - VT$ to the set of tree vertices VT.

- For each remaining vertex u in $V - VT$ that is connected to $u*$ by a shorter edge than the u's current distance label, update its labels by $u*$ and the weight of the edge between $u*$ and u, respectively.

ALGORITHM $Prim(G)$
  //Prim's algorithm for constructing a minimum spanning tree
  //Input: A weighted connected graph $G = \langle V, E \rangle$
  //Output: E_T, the set of edges composing a minimum spanning tree of G
  $V_T \leftarrow \{v_0\}$ //the set of tree vertices can be initialized with any vertex
  $E_T \leftarrow \varnothing$
  for $i \leftarrow 1$ **to** $|V| - 1$ **do**
      find a minimum-weight edge $e^* = (v^*, u^*)$ among all the edges (v, u)
      such that v is in V_T and u is in $V - V_T$
      $V_T \leftarrow V_T \cup \{u^*\}$
      $E_T \leftarrow E_T \cup \{e^*\}$
  return E_T
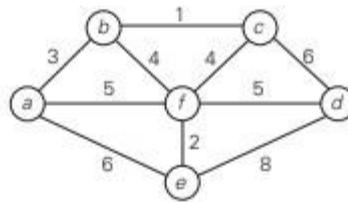
Tree vertices	Remaining vertices	Illustration
a(−, −)	**b(a, 3)** c(−, ∞) d(−, ∞) e(a, 6) f(a, 5)	
b(a, 3)	**c(b, 1)** d(−, ∞) e(a, 6) f(b, 4)	
c(b, 1)	d(c, 6) e(a, 6) **f(b, 4)**	
f(b, 4)	d(f, 5) **e(f, 2)**	
e(f, 2)	**d(f, 5)**	
d(f, 5)		

If a graph is represented by its adjacency lists and the priority queue is implemented as a min-heap, the running time of the algorithm is in $O(|E| \log |V|)$. This is because the algorithm performs $|V| - 1$ deletions of the smallest element and makes $|E|$ verifications and, possibly, changes of an element's priority in a min-heap of size not exceeding $|V|$. Each of these operations, as noted earlier, is a $O(\log |V|)$ operation. Hence, the running time of this implementation of Prim's algorithm is in

$$(|V| - 1 + |E|)O(\log |V|) = O(|E| \log |V|)$$

because, in a connected graph, $|V| - 1 \le |E|$.

Krushkal's algorithm

Kruskal's algorithm is a popular algorithm for finding the minimum spanning tree (MST) in a graph. An MST is a tree that spans all the vertices of a graph and has the minimum total edge weight among all possible spanning trees of the graph.

Kruskal's algorithm looks at a minimum spanning tree of a weighted connected graph G= {V, E} as an acyclic subgraph with |V| − 1 edges for which the sum of the edge weights is the smallest. the algorithm constructs a minimum spanning tree as an expanding sequence of subgraphs that are always acyclic but are not necessarily connected on the intermediate stages of the algorithm.

The algorithm begins by sorting the graph's edges in nondecreasing order of their weights. Then, starting with the empty subgraph, it scans this sorted list, adding the next edge on the list to the current subgraph if such an inclusion does not create a cycle and simply skipping the edge otherwise.

Kruskal's algorithm looks at a minimum spanning tree of a weighted connected graph G = (V, E) as an acyclic subgraph with |V| − 1 edges for which the sum of the edge weights is the smallest.

Kruskal's algorithm works as follows:

1. Sort all the edges of the graph in non-descending order based on their weights.
2. Start building the MST by adding edges one by one, taking care to avoid the creation of cycles in the tree.
3. To check if adding an edge creates a cycle, Kruskal's algorithm uses a disjoint-set data structure. This data structure keeps track of which vertices are connected and helps in detecting cycles.
4. The algorithm continues adding edges until all the vertices are connected and forms a tree.

Kruskal's algorithm is a greedy algorithm and is known to run in O(E log E) time, where E is the number of edges in the graph.

It is important to note that Kruskal's algorithm only works for undirected, connected, and non-cyclic graphs.

ALGORITHM *Kruskal(G)*

//Kruskal's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = \langle V, E \rangle$
//Output: E_T, the set of edges composing a minimum spanning tree of G
sort E in nondecreasing order of the edge weights $w(e_{i_1}) \leq \cdots \leq w(e_{i_{|E|}})$
$E_T \leftarrow \varnothing;$ *ecounter* $\leftarrow 0$ //initialize the set of tree edges and its size
$k \leftarrow 0$ //initialize the number of processed edges
while *ecounter* $< |V| - 1$ **do**
    $k \leftarrow k + 1$
    if $E_T \cup \{e_{i_k}\}$ is acyclic
        $E_T \leftarrow E_T \cup \{e_{i_k}\};$ *ecounter* $\leftarrow$ *ecounter* $+ 1$
return E_T
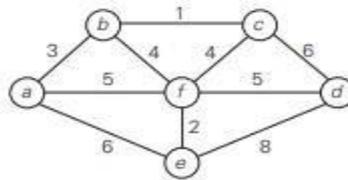
Tree edges	Sorted list of edges	Illustration
	bc ef ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
bc 1	**bc** ef ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
ef 2	bc **ef** ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
ab 3	bc ef **ab** bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
bf 4	bc ef ab **bf** cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
df 5	bc ef ab bf cf af **df** ae cd de 1 2 3 4 4 5 5 6 6 8	

Shortest path: Bellman-Ford algorithm

The Bellman-Ford algorithm is a widely used algorithm for finding the shortest path in a weighted graph. It's similar to Dijkstra's algorithm, but it's more versatile as it can handle graphs with negative edge weights, which Dijkstra's algorithm cannot handle.

The algorithm works by relaxing the edges of the graph one by one and updating the distance of each vertex from the source. A vertex's distance is relaxed if a new path to that vertex is found which is shorter than the previous one. This process is repeated |V|-1 times, where |V| is the number of vertices in the graph. After the relaxation process, if we still find any vertex witha distance that can be improved, this means that the graph has a negative weight cycle, and the algorithm returns an error.

The algorithm has a time complexity of $O(|V|*|E|)$, where |E| is the number of edges in the graph. This means that it's slower than Dijkstra's algorithm, which has a time complexity of $O(|E| + |V|\log|V|)$ when using a heap data structure. However, the Bellman-Ford algorithm is still widely used due to its versatility and its simplicity of implementation.

Input: A weighted graph G and a source node s.

Output: The shortest path from the source node s to all other nodes in the graph, or a report of the presence of negative weight cycles in the graph.

Set the distance of the source node s to itself as 0, and the distance of all other nodes as infinity.

For i from 1 to V-1, where V is the number of vertices in the graph:

For each edge (u, v) in the graph:

If the distance to the destination node v through edge (u, v) is shorter than the current distance to v, update the distance to v with the shorter distance.

Check for the presence of negative weight cycles:

For each edge (u, v) in the graph:

If the distance to the destination node v through edge (u, v) is shorter than the current distance to v, report the presence of a negative weight cycle in the graph.

If there is no negative weight cycle, return the distance of the shortest path from the source node s to all other nodes in the graph.

Warshall's algorithm

DEFINITION The *transitive closure* of a directed graph with n vertices can be defined as the $n \times n$ boolean matrix $T = \{t_{ij}\}$, in which the element in the ith row and the jth column is 1 if there exists a nontrivial path (i.e., directed path of a positive length) from the ith vertex to the jth vertex; otherwise, t_{ij} is 0.
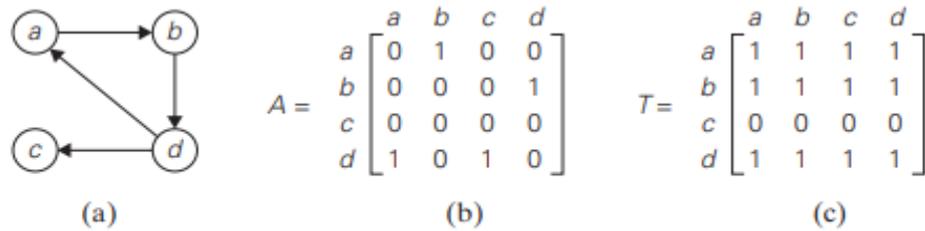
FIGURE 8.11 (a) Digraph. (b) Its adjacency matrix. (c) Its transitive closure.

3.2 WARSHALL'S AND FLOYD' ALGORITHM

Warshall's and Floyd's Algorithms: Warshall's algorithm for computing the transitive closure (there is a path between any two nodes) of a directed graph and Floyd's algorithm for the all-pairs shortest-paths problem. These algorithms are based on dynamic programming.

WARSHALL'S ALGORITHM (All-Pairs Path Existence Problem)

A **directed graph** (or digraph) is a graph, or set of vertices connected by edges, where the edges have a direction associated with them.

An **Adjacency matrix** A = {a_{ij}} of a directed graph is the boolean matrix that has 1 in its ith row and jth column if and only if there is a directed edge from the ith vertex to the jth vertex.

The **transitive closure** of a directed graph with n vertices can be defined as the n x n boolean matrix T = {t_{ij} }, in which the element in the ith row and the jth column is 1 if there exists a nontrivial path (i.e., directed path of a positive length) from the ith vertex to the jth vertex; otherwise, tij is 0.

Warshall's algorithm constructs the transitive closure through a series of n × n boolean matrices: $R^{(0)}, \ldots, R^{(k-1)}, R^{(k)}, \ldots R^{(n)}$.

The element $r_{ij}^{(k)}$ in the ith row and jth column of matrix $R^{(k)}$ (i, j = 1, 2, . . . , n, k = 0, 1, . . . , n) is equal to 1 if and only if there exists a directed path of a positive length from the ith vertex to the jth vertex with each intermediate vertex, if any, numbered not higher than k.

Steps to compute $R^{(0)}, \ldots, R^{(k-1)}, R^{(k)}, \ldots R^{(n)}$.

- The series starts with $R^{(0)}$, which does not allow any intermediate vertices in its paths; hence, $R^{(0)}$ is nothing other than the adjacency matrix of the digraph.
- $R^{(1)}$ contains the information about paths that can use the first vertex as intermediate. it may contain more 1's than $R^{(0)}$.
- The last matrix in the series, R(n), reflects paths that can use all n vertices of the digraph as intermediate and hence is nothing other than the digraph's transitive closure.
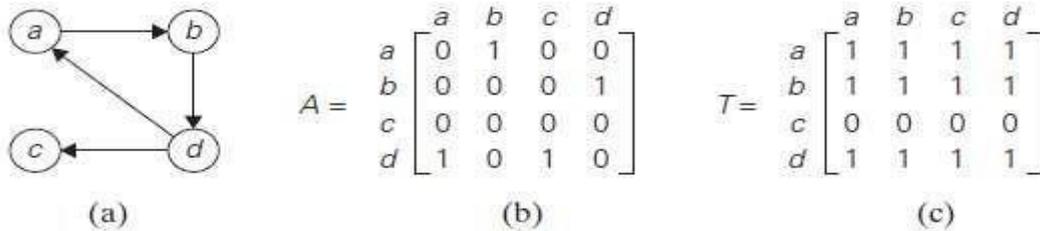
FIGURE 3.1 (a) Digraph. (b) Its adjacency matrix. (c) Its transitive closure.

All the elements of each matrix $R^{(k)}$ is computed from its immediate predecessor $R^{(k-1)}$. Let $r_{ij}^{(k)}$, the element in the ith row and jth column of matrix $R^{(k)}$, be equal to 1. This means that there exists a path from the ith vertex v_i to the jth vertex v_j with each intermediate vertex numbered not higher than k.

The transitive closure of a digraph can be generated with the help of depth-first search or breadth-first search. Every vertex as a starting point yields the transitive closure for all.

FIGURE 3.2 Rule for changing zeros in Warshall's algorithm.
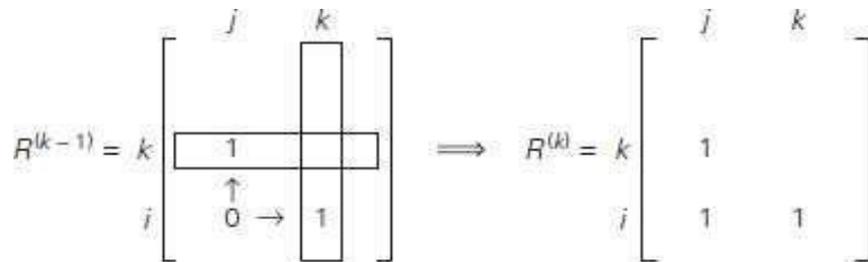
The first part of this representation means that there exists a path from v_i to v_k with each intermediate vertex numbered not higher than $k - 1$ (hence, $r_{ik}^{(k-1)} = 1$), and the second part means that there exists a path from v_k to v_j with each intermediate vertex numbered not higher than $k - 1$ (hence, $r_{kj}^{(k-1)} = 1$).

Thus the following formula generas the elements of matrix $R^{(k)}$ from the elements of matrix $R^{(k-1)}$:

$$r_{ij}^{(k)} = r_{ij}^{(k-1)} \quad \text{or} \quad \left(r_{ik}^{(k-1)} \text{ and } r_{kj}^{(k-1)} \right)$$

ApplyingWarshall's algorithm by hand:
- If an element r_{ij} is 1 in $R^{(k-1)}$, it remains 1 in $R^{(k)}$.
- If an element r_{ij} is 0 in $R^{(k-1)}$, it has to be changed to 1 in $R^{(k)}$ if and only if the element in its row i and column k and the element in its column j and row k are both 1's in$R^{(k-1)}$.

ALGORITHM Warshall(A[1..n, 1..n])
//ImplementsWarshall's algorithm for computing the transitive closure

//Input: The adjacency matrix A of a digraph with n vertices

//Output: The transitive closure of the digraph
$R^{(0)} \leftarrow A$

for k←1 **to** n **do**

      for i ←1 **to** n **do**

            for j ←1 **to** n **do**

                  $R^{(k)}[i, j] \leftarrow R^{(k-1)}[i, j]$ or $(R^{(k-1)}[i, k]$ **and** $R^{(k-1)}[k, j])$

return $R^{(n)}$
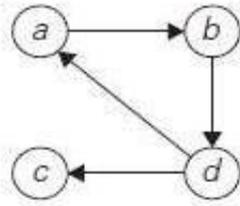
$R^{(0)} =$

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	0	1	0

1's reflect the existence of paths with no intermediate vertices ($R^{(0)}$ is just the adjacency matrix); boxed row and column are used for getting $R^{(1)}$.

$R^{(1)} =$

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	1	1	0

1's reflect the existence of paths with intermediate vertices numbered not higher than 1, i.e., just vertex **a** (note a new path from d to b); boxed row and column are used for getting $R^{(2)}$.

$R^{(2)} =$

	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	0	0	0	0
d	1	1	1	1

1's reflect the existence of paths with intermediate vertices numbered not higher than 2, i.e., **a** and **b** (note two new paths); boxed row and column are used for getting $R^{(3)}$.

$R^{(3)} =$

	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	0	0	0	0
d	1	1	1	1

1's reflect the existence of paths with intermediate vertices numbered not higher than 3, i.e., **a**, **b**, and **c** (no new paths); boxed row and column are used for getting $R^{(4)}$.

$R^{(4)} =$

	a	b	c	d
a	1	1	1	1
b	1	1	1	1
c	0	0	0	0
d	1	1	1	1

1's reflect the existence of paths with intermediate vertices numbered not higher than 4, i.e., **a**, **b**, **c**, and **d** (note five new paths).

FLOYD'S ALGORITHM (All-Pairs Shortest-Paths Problem)

Floyd'sl algorithm is an algorithm for finding shortest paths for all pairs in a connected graph (undirected or directed) with (+/-) edge weights.
weighted

A **distance matrix** is a matrix (two-dimensional array) containing the distances, taken pairwise, between the vertices of graph.

The lengths of shortest paths in an n × n matrix D called the distance matrix: the element d_{ij} in the ith row and the jth column of this matrix indicates the length of the shortest path from the ith vertex to the jth vertex.

We can generate the distance matrix with an algorithm that is very similar to algorithm is called Floyd's algorithm.
Warshall's

Floyd's algorithm computes the distance matrix of a weighted graph with n vertices through a series of $n \times n$ matrices:

$$D^{(0)}, \ldots, D^{(k-1)}, D^{(k)}, \ldots, D^{(n)}.$$
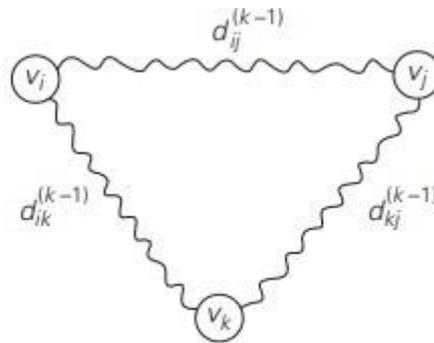
Underlying idea of Floyd's algorithm.

$$d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)}, \ d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\} \quad \text{for } k \geq 1, \ d_{ij}^{(0)} = w_{ij}.$$

ALGORITHM *Floyd*$(W[1..n, 1..n])$

//Implements Floyd's algorithm for the all-pairs shortest-paths problem
//Input: The weight matrix W of a graph with no negative-length cycle
//Output: The distance matrix of the shortest paths' lengths
$D \leftarrow W$ //is not necessary if W can be overwritten
for $k \leftarrow 1$ **to** n **do**
    for $i \leftarrow 1$ **to** n **do**
        for $j \leftarrow 1$ **to** n **do**
            $D[i, j] \leftarrow \min\{D[i, j], D[i, k] + D[k, j]\}$
    return D
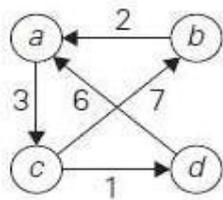
$D^{(0)} =$

	a	b	c	d
a	0	∞	3	∞
b	2	0	∞	∞
c	∞	7	0	1
d	6	∞	∞	0

Lengths of the shortest paths with no intermediate vertices ($D^{(0)}$ is simply the weight matrix).

$D^{(1)} =$

	a	b	c	d
a	0	∞	3	∞
b	2	0	**5**	∞
c	∞	7	0	1
d	6	∞	**9**	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 1, i.e., just *a* (note two new shortest paths from *b* to *c* and from *d* to *c*).

$D^{(2)} =$

	a	b	c	d
a	0	∞	3	∞
b	2	0	5	∞
c	**9**	7	0	1
d	6	∞	9	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 2, i.e., **a** and **b** (note a new shortest path from *c* to *a*).

$D^{(3)} =$

	a	b	c	d
a	0	**10**	3	**4**
b	2	0	5	**6**
c	9	7	0	1
d	6	**16**	9	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 3, i.e., **a**, **b**, and **c** (note four new shortest paths from *a* to *b*, from a to *d*, from *b* to *d*, and from *d* to *b*).

$D^{(4)} =$

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	**7**	7	0	1
d	6	16	9	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 4, i.e., **a**, **b**, **c**, and **d** (note a new shortest path from *c* to *a*).

FIGURE 3.5 Application of Floyd's algorithm to the digraph shown. Updated elements are shown in bold.

Dijikstra's Algorithm:

The single-source shortest-paths problem: for a given vertex called the source in a weighted connected graph, find shortest paths to all its other vertices.

The single-source shortest-paths problem asks for a family of paths, each leading from the source to a different vertex in the graph, though some paths may, of course, have edges in common. The best-known algorithm for the single-source shortest-paths problem, called Dijkstra's algorithm.
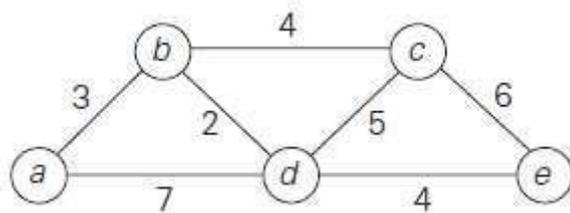
ALGORITHM *Dijkstra(G, s)*

      //Dijkstra's algorithm for single-source shortest paths
      //Input: A weighted connected graph $G = (V, E)$ with nonnegative weights and its vertex s
      //Output: The length dv of a shortest path from s to v and its penultimate vertex pv for every
      //   vertex v in V
Initialize(Q) //initialize priority queue to empty
for every vertex v in V
      $dv \leftarrow \infty$; $pv \leftarrow$ **null**
      Insert (Q, v, dv) //initialize vertex priority in the priority queue
$Ds \leftarrow 0$; *Decrease(Q, s, d_s)* //update priority of s with d_s
$V_T \leftarrow \Phi$
for $i \leftarrow 0$ **to** $|V| - 1$ **do**
      $u^* \leftarrow$ *DeleteMin(Q)* //delete the minimum priority element
      $V_T \leftarrow V_T \cup \{u^*\}$
      for every vertex u in $V - VT$ that is adjacent to u^* **do**
            if $d_u{}^* + w(u^*, u) < d_u$
            $d_u \leftarrow d_u{}^* + w(u^*, u)$; $p_u \leftarrow u^*$
            Decrease(Q, u, d_u)

The time efficiency of Dijkstra's algorithm depends on the data structures used for implementing the priority queue and for representing an input graph itself. It is in Θ ($|V|^2$) for graphs represented by their weight matrix and the priority queue implemented as an unordered array. For graphs represented by their adjacency lists and the priority queue implemented as a min-heap, it is in $O(|E| \log |V|)$.

Tree vertices	Remaining vertices	Illustration
a(−, 0)	**b(a, 3)** c(−, ∞) d(a, 7) e(−, ∞)	
b(a, 3)	c(b, 3 + 4) **d(b, 3 + 2)** e(−, ∞)	
d(b, 5)	**c(b, 7)** e(d, 5 + 4)	
c(b, 7)	**e(d, 9)**	
e(d, 9)		

FIGURE 3.16 Application of Dijkstra's algorithm. The next closest vertex is shown in bold

From a to b : a − b of length 3

From a to c : a − b − c of length 7

From a to d : a − b− d of length 5

From a to e : a − b− d − e of length 9

The shortest paths (identified by following nonnumeric labels backward from a destination vertex in the left column to the source) and their lengths (given by numeric labels of the tree vertices) are as follows:

Maximum Flow Problem

Problem of maximizing the flow of a material through a transportation network (e.g., pipeline system, communications or transportation networks)

Formally represented by a connected weighted digraph with n vertices numbered from 1 to n with the following properties:

- Contains exactly one vertex with no entering edges, called the *source* (numbered 1)
- Contains exactly one vertex with no leaving edges, called the *sink* (numbered n)
- Has positive integer weight u_{ij} on each directed edge (i,j), called the *edge capacity*, indicating the upper bound on the amount of the material that can be sent from i to j through this edge.
- A digraph satisfying these properties is called a **flow network** or simply a network.

Example of Flow Network
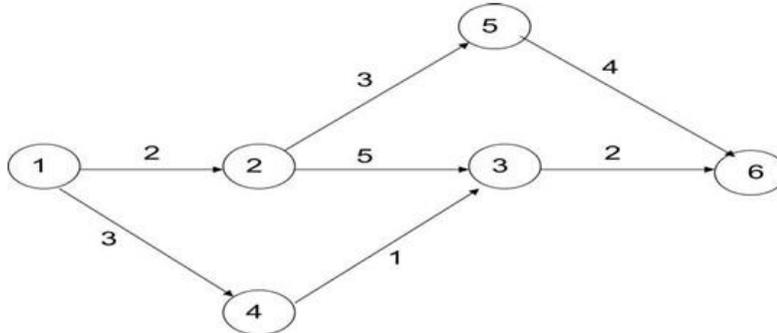
Node (1) = source
Node(6) = sink

Definition of a Flow

A *flow* is an assignment of real numbers x_{ij} to edges (i,j) of a given network that satisfy the following:

- *flow-conservation requirements*

    The total amount of material entering an intermediate vertex must be equal to the total amount of the material leaving the vertex

- *capacity constraints*

    $$0 \leq x_{ij} \leq u_{ij} \text{ for every edge } (i,j) \in E$$

Flow value and Maximum Flow Problem

Since no material can be lost or added to by going through intermediate vertices of the network, the total amount of the material leaving the source must end up at the sink:

$$\sum_{j:\ (1,j)\ \in\ E} x_{1j} \quad - \quad \sum_{j:\ (j,n)\ \in\ E} x_{jn}$$

The *value* of the flow is defined as the total outflow from the source (= the total inflow into the sink). The *maximum flow problem* is to find a flow of the largest value (maximum flow) for a given network.

Maximum-Flow Problem as LP problem

Maximize $v = \sum x_{1j}$

$\quad\quad\quad\quad j: (1,j) \in E$

subject to

$$\sum x_{ji} - \sum x_{ij} = 0 \quad \text{for } i = 2, 3, \ldots, n\text{-}1$$

$j: (j,i) \in E \quad j: (i,j) \in E$

$0 \leq x_{ij} \leq u_{ij}$ for every edge $(i,j) \in E$

Augmenting Path (Ford-Fulkerson) Method

- Start with the zero flow ($x_{ij} = 0$ for every edge).
- On each iteration, try to find a *flow-augmenting path* from source to sink, which a path along which some additional flow can be sent.
- If a flow-augmenting path is found, adjust the flow along the edges of this path to get a flow of increased value and try again.
- If no flow-augmenting path is found, the current flow is maximum.

Example 1
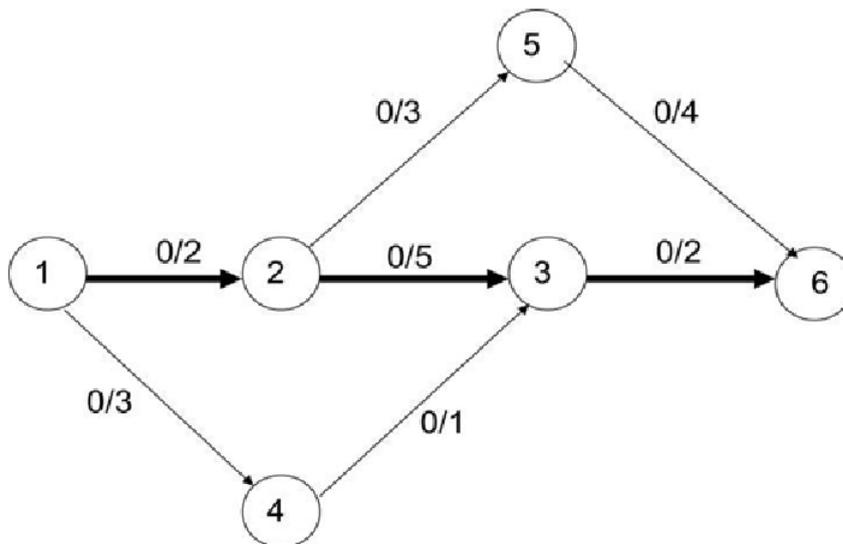
Augmenting path: $1 \rightarrow 2 \rightarrow 3 \rightarrow 6$

x_{ij}/u_{ij}
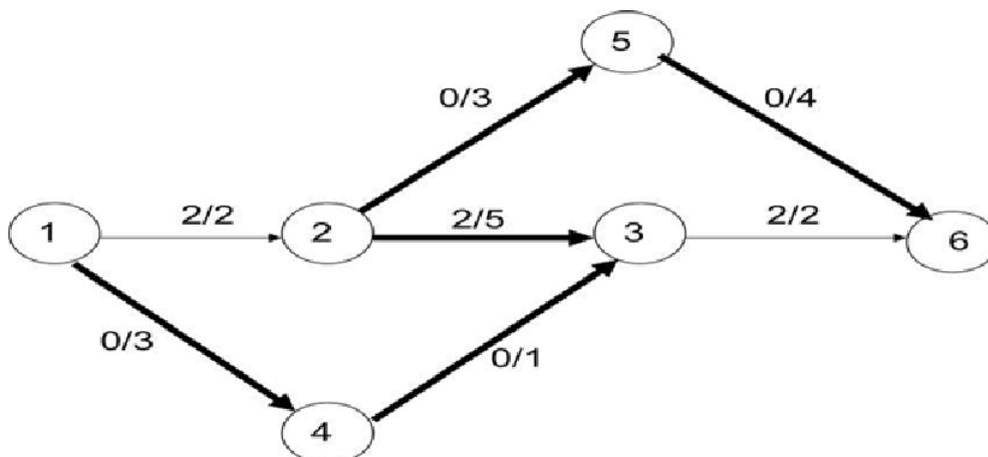
Augmenting path: $1 \rightarrow 4 \rightarrow 3 \leftarrow 2 \rightarrow 5 \rightarrow 6$
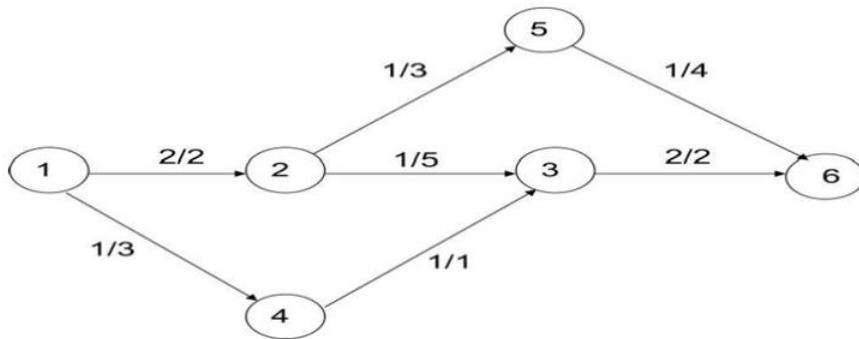
Finding a flow-augmenting path

To find a flow-augmenting path for a flow x, consider paths from source to sink in the underlying underlined graph in which any two consecutive vertices i, j are either:

- connected by a directed edge (i to j) with some positive unused capacity $r_{ij} = u_{ij} - x_{ij}$
  - known as *forward edge* ($\rightarrow$)

  OR

- connected by a directed edge (j to i) with positive flow x_{ji}
  - known as *backward edge* ($\leftarrow$)

If a flow-augmenting path is found, the current flow can be increased by r units by increasing x_{ij} by r on each forward edge and decreasing x_{ji} by r on each backward edge, where

r = min {r_{ij} on all forward edges, x_{ji} on all backward edges}

r = min {r_{ij} on all forward edges, x_{ji} on all backward edges}

- Assuming the edge capacities are integers, r is a positive integer
- On each iteration, the flow value increases by at least 1
- Maximum value is bounded by the sum of the capacities of the edges leaving the source; hence the augmenting-path method has to stop after a finite number of iterations
- The final flow is always maximum, its value doesn't depend on a sequence of augmenting paths used
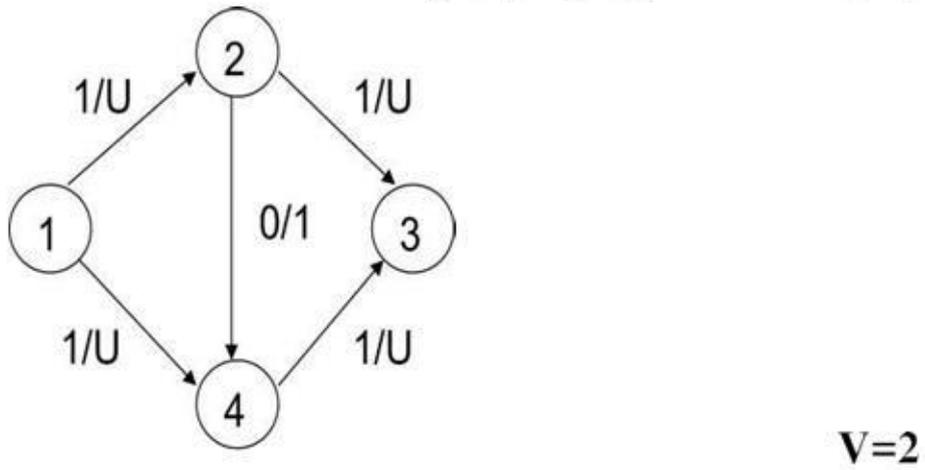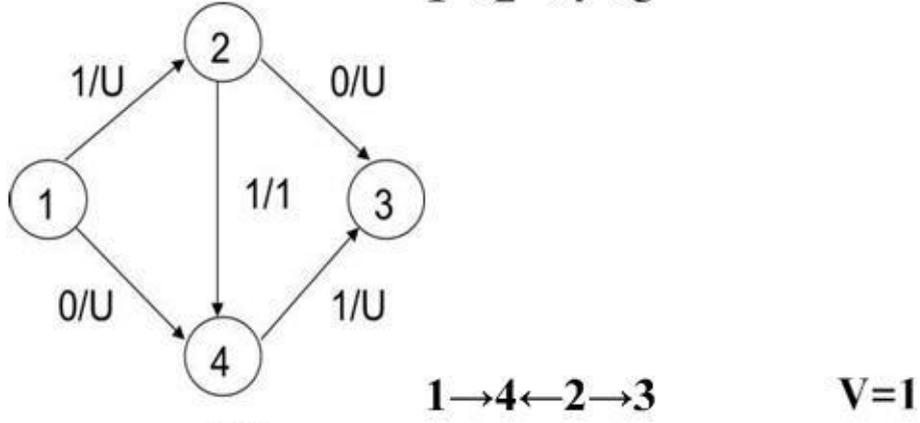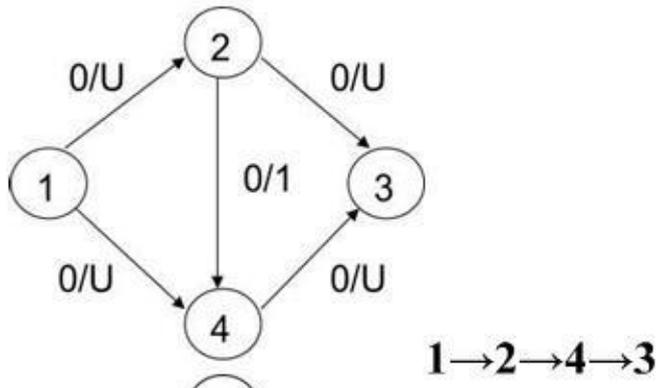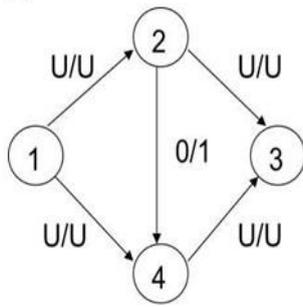
Performance degeneration of the method

- The augmenting-path method doesn't prescribe a specific way for generating flow-augmenting paths
- Selecting a bad sequence of augmenting paths could impact the method's efficiency

Example

$1 \rightarrow 2 \rightarrow 4 \rightarrow 3$

$1 \rightarrow 4 \leftarrow 2 \rightarrow 3$ V=1

V=2

...
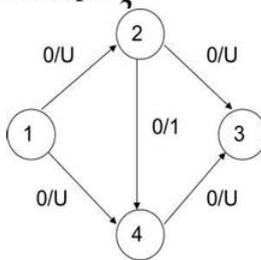
V=2U

Requires 2U iterations to reach maximum flow of value 2U
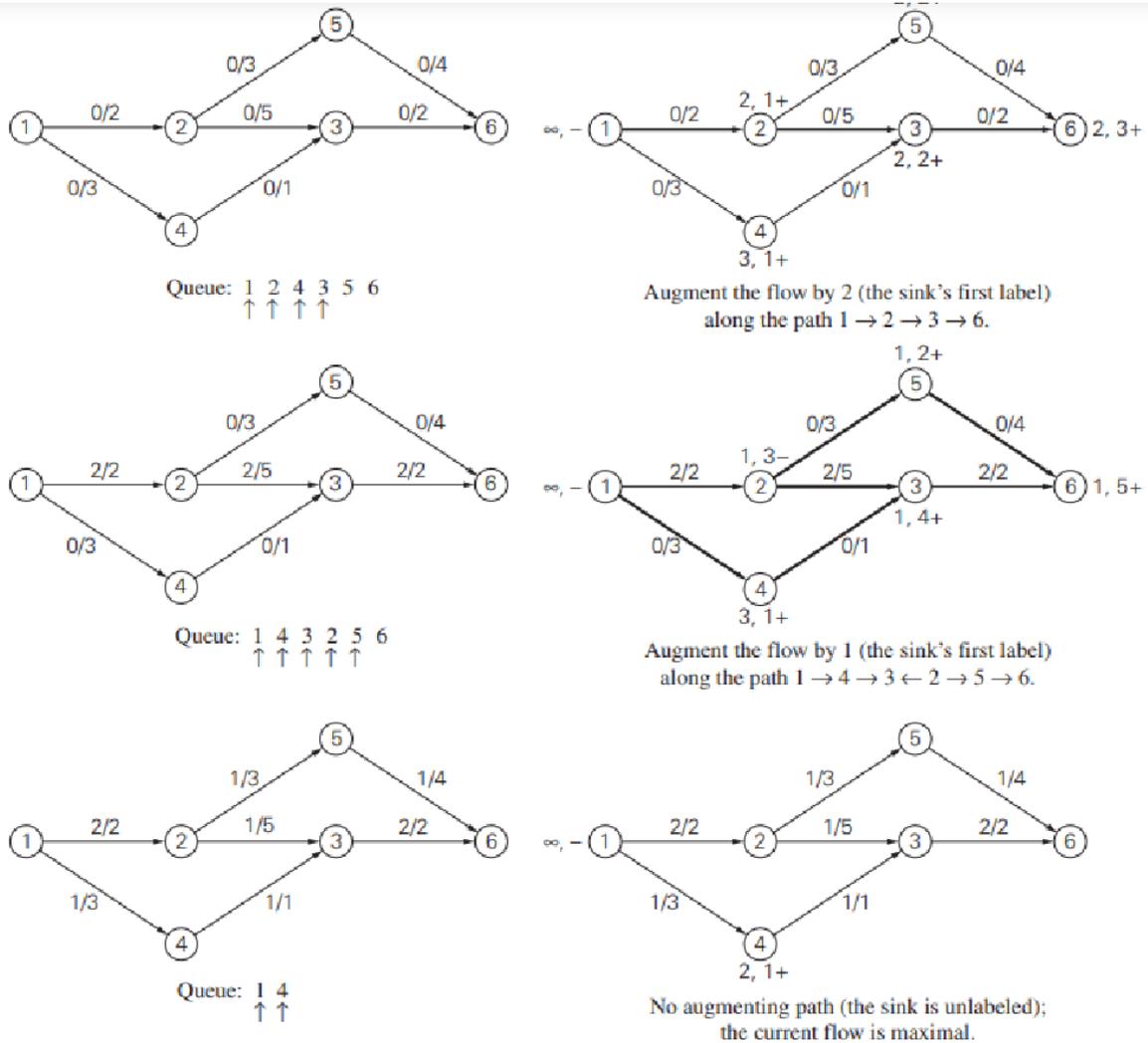
Shortest-Augmenting-Path Algorithm

Generate augmenting path with the least number of edges by BFS as follows.

Starting at the source, perform BFS traversal by marking new (unlabeled) vertices with two labels:

- first label – indicates the amount of additional flow that can be brought from the source to the vertex being labeled

- second label – indicates the vertex from which the vertex being labeled was reached, with "+" or "–" added to the second label to indicate whether the vertex was reached via a forward or backward edge

Vertex labeling

- The source is always labeled with ∞,-
- All other vertices are labeled as follows:
  - If unlabeled vertex j is connected to the front vertex i of the traversal queue by a directed edge from i to j with positive unused capacity $r_{ij} = u_{ij} - x_{ij}$ (forward edge), vertex j is labeled with l_j, i^+, where $l_j = \min\{l_i, r_{ij}\}$

  - If unlabeled vertex j is connected to the front vertex i of the traversal queue by a directed edge from j to i with positive flow x_{ji} (backward edge), vertex j is labeled l_j, i^-, where $l_j = \min\{l_i, x_{ji}\}$
- If the sink ends up being labeled, the current flow can be augmented by the amount indicated by the sink's first label.
- The augmentation of the current flow is performed along the augmenting path traced by following the vertex second labels from sink to source; the current flow quantities are increased on the forward edges and decreased on the backward edges of this path.
- If the sink remains unlabeled after the traversal queue becomes empty, the algorithm returns the current flow as maximum and stops.
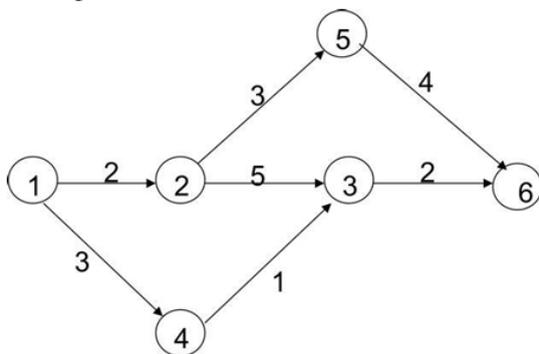
Queue: 1 2 4 3 5 6
↑ ↑ ↑ ↑

Augment the flow by 2 (the sink's first label)
along the path 1 → 2 → 3 → 6.

Queue: 1 4 3 2 5 6
↑ ↑ ↑ ↑ ↑

Augment the flow by 1 (the sink's first label)
along the path 1 → 4 → 3 ← 2 → 5 → 6.

Queue: 1 4
↑ ↑

No augmenting path (the sink is unlabeled);
the current flow is maximal.

Definition of a Cut

Let X be a set of vertices in a network that includes its source but does not include its sink, and let X, the complement of X, be the rest of the vertices including the sink. The *cut* induced by this partition of the vertices is the set of all the edges with a tail in X and a head in X.

Capacity of a cut is defined as the sum of capacities of the edges that compose the cut.

- We'll denote a cut and its capacity by C(X,X) and c(X,X)
- Note that if all the edges of a cut were deleted from the network, there would be no directed path from source to sink
- *Minimum cut* is a cut of the smallest capacity in a given network

Examples of network cuts

If X = {1} and X = {2,3,4,5,6}, C(X,X) = {(1,2), (1,4)}, c = 5

If X ={1,2,3,4,5} and X = {6}, C(X,X) = {(3,6), (5,6)}, c = 6

If X = {1,2,4} and X ={3,5,6}, C(X,X) = {(2,3), (2,5), (4,3)}, c = 9

Max-Flow Min-Cut Theorem

1. The value of maximum flow in a network is equal to the capacity of its minimum cut
2. The shortest augmenting path algorithm yields both a maximum flow and a minimum cut:
   - Maximum flow is the final flow produced by the algorithm
   - Minimum cut is formed by all the edges from the labeled vertices to unlabeled vertices on the last iteration of the algorithm.
   - All the edges from the labeled to unlabeled vertices are full, i.e., their flow amounts are equal to the edge capacities, while all the edges from the unlabeled to labeled vertices, if any, have zero flow amounts on them.

ALGORITHM *ShortestAugmentingPath(G)*

//Implements the shortest-augmenting-path algorithm

//Input: A network with single source 1, single sink n, and positive integer capacities u_{ij} on
//   its edges (i, j)

//Output: A maximum flow x

assign $x_{ij} = 0$ to every edge (i, j) in the network
label the source with ∞, $-$ and add the source to the empty queue Q

while not *Empty(Q)* **do**
  $i \leftarrow Front(Q); Dequeue(Q)$
  for every edge from i to j **do** //forward edges
    if j is unlabeled

      $r_{ij} \leftarrow u_{ij} - x_{ij}$
      if $r_{ij} > 0$

        $l_j \leftarrow \min\{l_i, r_{ij}\}$; label j with l_j, $i +$
        $Enqueue(Q, j)$
  for every edge from j to i **do** //backward edges
    if j is unlabeled

      if $x_{ji} > 0$

        $l_j \leftarrow \min\{l_i, x_{ji}\}$; label j with l_j, $i-$
        $Enqueue(Q, j)$
  if the sink has been labeled

    //augment along the augmenting path found
    $j \leftarrow n$ //start at the sink and move backwards using second labels

    while $j \neq 1$ //the source hasn't been reached
      if the second label of vertex j is $i+$

        $x_{ij} \leftarrow x_{ij} + l_n$
      else //the second label of vertex j is $i-$

        $x_{ij} \leftarrow x_{ij} - l_n$
      $j \leftarrow i; i \leftarrow$ the vertex indicated by i's second label

    erase all vertex labels except the ones of the source
    reinitialize Q with the source

 **return** x //the current flow is maximum

Time Efficiency

- The number of augmenting paths needed by the shortest-augmenting-path algorithm never exceeds $nm/2$, where n and m are the number of vertices and edges, respectively.
- Since the time required to find shortest augmenting path by breadth-first search is in $O(n+m)=O(m)$ for networks represented by their adjacency lists, the time efficiency of the shortest-augmenting-path algorithm is in $O(nm^2)$ for this representation.
- More efficient algorithms have been found that can run in close to $O(nm)$ time, but these algorithms don't fall into the iterative-improvement paradigm.

Maximum Matching in Bipartite

Bipartite Graphs

Bipartite graph: a graph whose vertices can be partitioned into two disjoint sets V and U, not necessarily of the same size, so that every edge connects a vertex in V to a vertex in U.

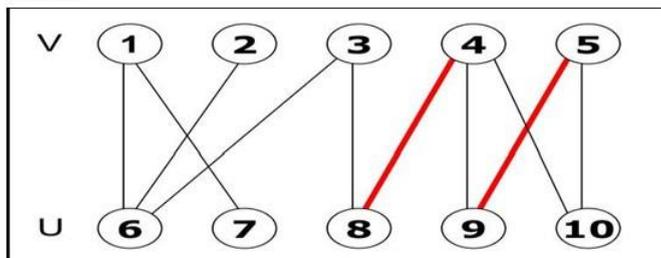A graph is bipartite if and only if it does not have a cycle of an odd length.
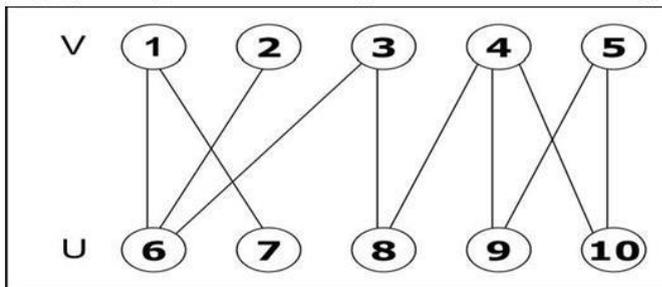
A bipartite graph is *2-colorable*: the vertices can be colored in two colors so that every edge has its vertices colored differently
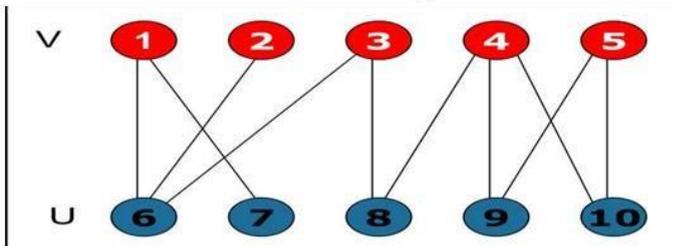
Graph

Matching in a Graph

A *matching* in a graph is a subset of its edges with the property that no two edges share a vertex

a matching in this graph M = {(4,8), (5,9)}

A *maximum* (or *maximum cardinality*) *matching* is a matching with the largest number of edges
- always exists
- not always unique

Free Vertices and Maximum Matching

For a given matching M, a vertex is called *free* (or *unmatched*) if it is not an end point of any edge in M; otherwise, a vertex is said to be *matched*

- If every vertex is matched, then M is a maximum matching
- If there are unmatched or free vertices, then M may be able to be improved
- We can immediately increase a **matching** by adding an edge connecting two free vertices (e.g., (1,6) above)
- Matched vertex = 4, 5, 8, 9. Free vertex = 1, 2, 3, 6, 7, 10.

Augmenting Paths and Augmentation

An *augmenting path* for a matching M is a path from a free vertex in V to a free vertex in U whose edges alternate between edges not in M and edges in M
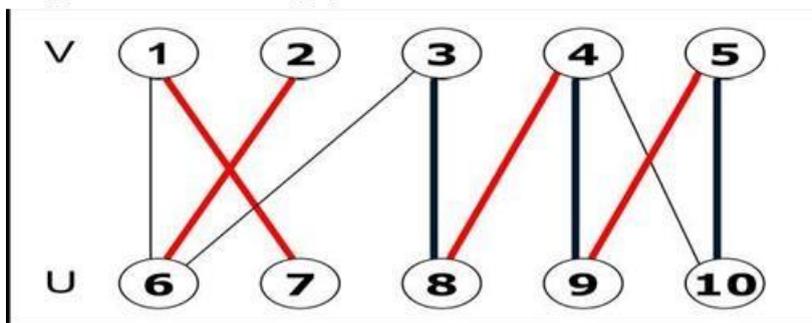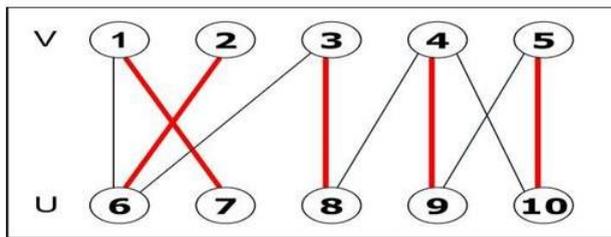
- The length of an augmenting path is always odd
- Adding to M the odd numbered path edges and deleting from it the even numbered path edges increases the matching size by 1 (*augmentation*)
- One-edge path between two free vertices is special case of augmenting path
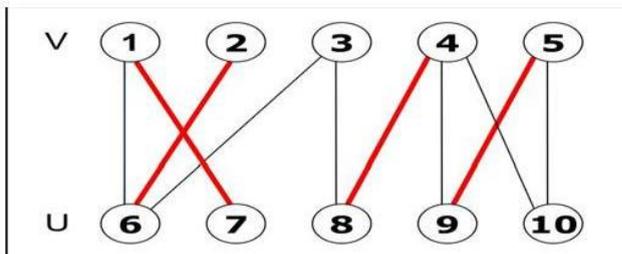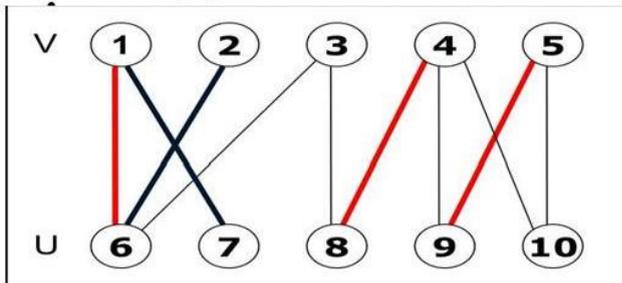
Augmentation along path 2,6,1,7

Augmentation along 3, 8, 4, 9, 5, 10

Matching on the right is maximum (*perfect* matching).

Theorem: A matching M is maximum if and only if there exists no augmenting path with respect to M.

Augmenting Path Method (template)

- Start with some initial matching . e.g., the empty set
- Find an augmenting path and augment the current matching along that path. e.g., using breadth-first search like method
- When no augmenting path can be found, terminate and return the last matching, which is maximum

ALGORITHM *MaximumBipartiteMatching(G)*

//Finds a maximum matching in a bipartite graph by a BFS-like traversal
//Input: A bipartite graph $G = \langle V, U, E \rangle$
//Output: A maximum-cardinality matching M in the input graph
initialize set M of edges with some valid matching (e.g., the empty set)
initialize queue Q with all the free vertices in V (in any order)
while not *Empty(Q)* **do**
$\quad w \leftarrow Front(Q); \quad Dequeue(Q)$
$\quad$ **if** $w \in V$
$\quad\quad$ **for** every vertex u adjacent to w **do**
$\quad\quad\quad$ **if** u is free
$\quad\quad\quad\quad$ //augment
$\quad\quad\quad\quad M \leftarrow M \cup (w, u)$
$\quad\quad\quad\quad v \leftarrow w$
$\quad\quad\quad\quad$ **while** v is labeled **do**
$\quad\quad\quad\quad\quad u \leftarrow$ vertex indicated by v's label; $\quad M \leftarrow M - (v, u)$
$\quad\quad\quad\quad\quad v \leftarrow$ vertex indicated by u's label; $\quad M \leftarrow M \cup (v, u)$
$\quad\quad\quad\quad$ remove all vertex labels
$\quad\quad\quad\quad$ reinitialize Q with all free vertices in V
$\quad\quad\quad\quad$ **break** //exit the for loop
$\quad\quad\quad$ **else** //u is matched
$\quad\quad\quad\quad$ **if** $(w, u) \notin M$ **and** u is unlabeled
$\quad\quad\quad\quad\quad$ label u with w
$\quad\quad\quad\quad\quad Enqueue(Q, u)$
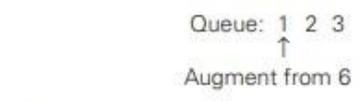$\quad$ **else** //$w \in U$ (and matched)
$\quad\quad$ label the mate v of w with w
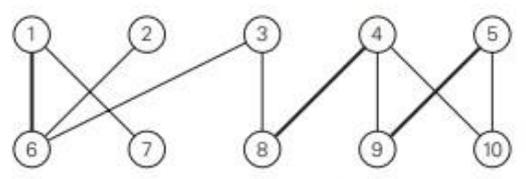$\quad\quad Enqueue(Q, v)$
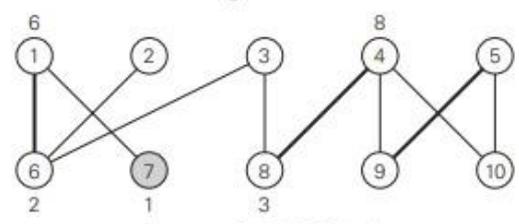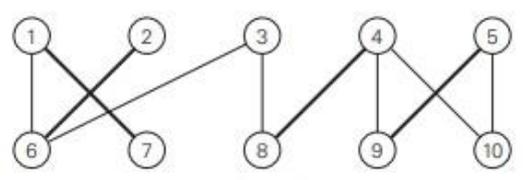**return** M //current matching is maximum
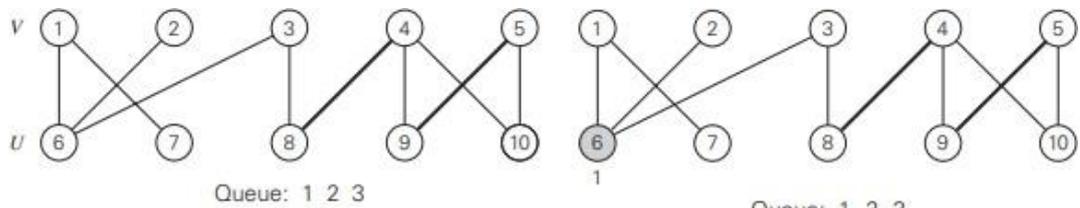
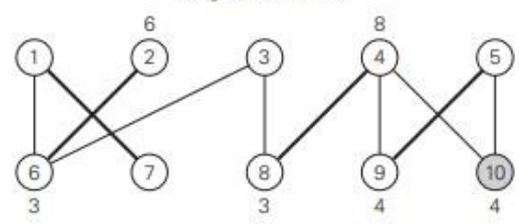Queue: 1 2 3
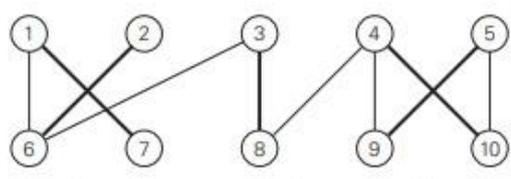
Queue: 1 2 3
↑
Augment from 6

Queue: 2 3

Queue: 2 3 6 8 1 4
↑ ↑ ↑ ↑ ↑
Augment from 7

Queue: 3

Queue: 3 6 8 2 4 9
↑ ↑ ↑ ↑ ↑
Augment from 10

Queue: empty ⇒ maximum matching

`UNIT III ALGORITHM DESIGN TECHNIQUES

Divide and Conquer methodology: Finding maximum and minimum - Merge sort - Quick sort

Dynamic programming: Elements of dynamic programming — Matrix-chain multiplication - Multi stage graph —Optimal Binary Search Trees.

Greedy Technique: Elements of the greedy strategy - Activity-selection problem –- Optimal Merge pattern — Huffman Trees.

DIVIDE AND CONQUER APPROACHES

Divide-and-conquer is probably the best-known general algorithm design technique. Divide-and-conquer algorithms work according to the following general plan:

1. A problem is divided into several subproblems of the same type, ideally of about equal size.

2. The subproblems are solved.

3. If necessary, the solutions to the subproblems are combined to get a solution to the original problem. As an example, let us consider the problem of computing the sum of n numbers $a_0, \ldots, a_{n-1}$. If $n > 1$, we can divide the problem into two instances of the same problem: to compute the sum of the first $n/2$ numbers and to compute the sum of the remaining $n/2$ numbers.

Once each of these two sums is computed by applying the same method recursively, we can add their values to get the sum in question:

$$a_0 + \cdots + a_{n-1} = (a_0 + \cdots + a_{n/2-1}) + (a_{n/2} + \cdots + a_{n-1}).$$
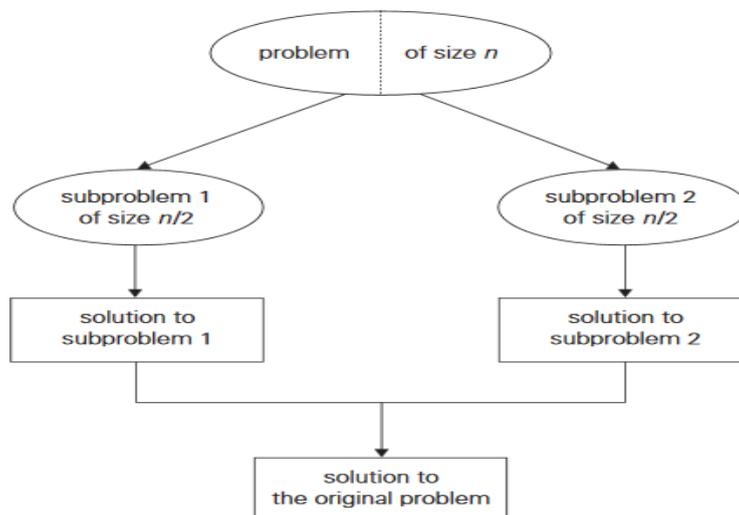
As mentioned above, in the most typical case of divide-and-conquer a problem's instance of size n is divided into two instances of size $n/2$. More generally, an instance of size n can be divided into b instances of size n/b, with a of them needing to be solved. (Here, a and b are constants; $a \geq 1$ and $b > 1$.)

Assuming that size n is a power of b to simplify our analysis, we get the following recurrence for the running time $T(n)$:

$$T(n) = aT(n/b) + f(n), \qquad\qquad (a)$$

where $f(n)$ is a function that accounts for the time spent on dividing an instance of size n into instances of size n/b and combining their solutions. (For the sum example above, $a = b = 2$ and $f(n) = 1$.) Recurrence (a) is called the **general divide-and-conquer recurrence**. Obviously, the order of growth of its solution $T(n)$ depends on the values of the constants a and b and the order of growth of the function $f(n)$.

Master Theorem If $f(n) \in \Theta(n^d)$ where $d \geq 0$ in recurrence (5.1), then

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d, \\ \Theta(n^d \log n) & \text{if } a = b^d, \\ \Theta(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

Analogous results hold for the O and Ω notations, too.

For example, the recurrence for the number of additions $A(n)$ made by the divide-and-conquer sum-computation algorithm (see above) on inputs of size $n = 2^k$ is

$$A(n) = 2A(n/2) + 1.$$

Thus, for this example, $a = 2$, $b = 2$, and $d = 0$; hence, since $a > b^d$,

$$A(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n).$$

<u>Finding maximum and minimum:</u>

Divide and conquer is a common algorithmic technique that can be used to find the maximum and minimum elements in an array. The general idea is to split the array into smaller sub-arrays, recursively find the maximum and minimum values in each sub-array, and then combine the results to obtain the maximum and minimum values for the original array.

Here is a general outline of how the divide and conquer algorithm works:

1. Divide the array into two equal halves.

2. Recursively find the maximum and minimum values in each half.

3. Compare the maximum and minimum values from each half to obtain the overall maximum and minimum values for the entire array.

Here is the pseudo-code for the algorithm:

function find_max_min(arr, left, right):

  if left == right:

    # Only one element in the array

    return arr[left], arr[left]

  elif left == right - 1:

    # Two elements in the array

    return max(arr[left], arr[right]), min(arr[left], arr[right])

  else:

    # More than two elements in the array

    mid = (left + right) // 2

    max_left, min_left = find_max_min(arr, left, mid)

    max_right, min_right = find_max_min(arr, mid+1, right)

    return max(max_left, max_right), min(min_left, min_right)

In this code, the function find_max_min takes an array arr, a left index left, and a right index right. If there is only one element in the array (i.e., left == right), the function returns that element as both the maximum and minimum. If there are two elements in the array (i.e., left == right - 1), the function compares the two elements and returns the maximum and minimum values. If there are more than two elements in the array, the function recursively calls itself on the left and right halves of the array and combines the results to obtain the maximum and minimum values for the entire array.

The time complexity of this algorithm is O(n log n), where n is the size of the array. This is because the algorithm recursively divides the array in half log n times and performs constant time operationson each half. Therefore, the total number of operations is proportional to n log n.

Mergesort

Mergesort is a perfect example of a successful application of the divide-and- conquer technique. It sorts a given array $A[0..n-1]$ by dividing it into two halves $A[0.. n/2-1]$ and $A[n/2..n-1]$, sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one.

ALGORITHM *Mergesort(A[0..n − 1])*

//Sorts array $A[0..n-1]$ by recursive mergesort

        //Input: An array $A[0..n-1]$ of orderable elements

        //Output: Array $A[0..n-1]$ sorted in nondecreasing order

        if $n > 1$

           copy $A[0.. n/2-1]$ to $B[0.. n/2-1]$

           copy $A[n/2 ..n-1]$ to $C[0.. n/2-1]$

           Mergesort(B [0.. n/2 − 1])

           Mergesort(C[0.. n/2 − 1])

           Merge(B, C, A) //see below

The ***merging*** of two sorted arrays can be done as follows. Two pointers (array indices) are initialized to point to the first elements of the arrays being merged. The elements pointed to are compared, and the smaller of them is added to a new array being constructed; after that, the index of the smaller element is incremented to point to its immediate successor in the array it was copied from. This operation is repeated until one of the two given arrays is exhausted, and then the remaining elements of the other array are copied to the end of the new array.

ALGORITHM *Merge(B [0..p − 1], C[0..q − 1], A[0..p +q − 1])*

        //Merges two sorted arrays into one sorted array

        //Input: Arrays $B[0..p-1]$ and $C[0..q-1]$ both sorted

        //Output: Sorted array $A[0..p +q-1]$ of the elements of B and C

        $i \leftarrow 0; j \leftarrow 0; k \leftarrow 0$

while $i < p$ **and** $j < q$ **do**

if $B[i] \leq C[j]$

$\qquad A[k] \leftarrow B[i];$

$\qquad i \leftarrow i + 1$

$\qquad$ **else** $A[k] \leftarrow C[j];$

$\qquad j \leftarrow j + 1$

$\qquad k \leftarrow k + 1$

if $i = p$

$\qquad$ copy $C[j..q - 1]$ to $A[k..p + q - 1]$

$$\underline{\textbf{else}}$$

copy $B[i..p - 1]$ to $A[k..p + q - 1]$

The operation of the algorithm on the list 8, 3, 2, 9, 7, 1, 5, 4 is illustrated below
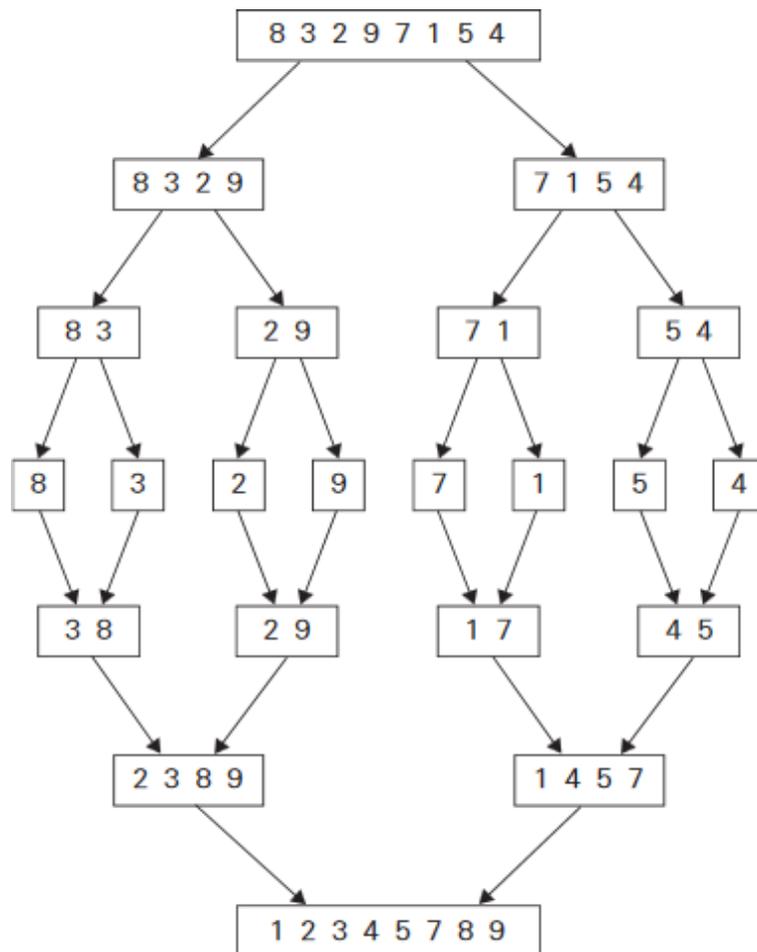
How efficient is mergesort? Assuming for simplicity that n is a power of 2, the recurrence relation for the number of key comparisons $C(n)$ is

$$C(n) = 2C(n/2) + C_{merge}(n) \quad \text{for } n > 1, \ C(1) = 0.$$

Therefore, for the worst case, $C_{merge}(n) = n - 1$, and we have the recurrence

$$C_{wor\,st}(n) = 2C_{wor\,st}(n/2) + n - 1 \quad \text{for } n > 1, \ C_{wor\,st}(1) = 0$$

Hence, according to the Master Theorem, $C_{worst}(n) \in \Theta(n \log n)$ (why?). In fact, it is easy to find the exact solution to the worst-case recurrence for $n = 2^k$:

$$C_{worst}(n) = n \log_2 n - n + 1.$$

Quicksort

Quicksort is the other important sorting algorithm that is based on the divide-and-conquer approach. Unlike mergesort, which divides its input elements according to their position in the array, quicksort divides them according to their value.

A partition is an arrangement of the array's elements so that all the elements to the left of some element $A[s]$ are less than or equal to $A[s]$, and all the elements to the right of $A[s]$ are greater than or equal to it:

$$\underbrace{A[0] \ldots A[s-1]}_{\text{all are } \leq A[s]} \ A[s] \ \underbrace{A[s+1] \ldots A[n-1]}_{\text{all are } \geq A[s]}$$

Obviously, after a partition is achieved, $A[s]$ will be in its final position in the sorted array, and we can continue sorting the two subarrays to the left and to the right of $A[s]$ independently (e.g., by the same method). Note the difference with mergesort: there, the division of the problem into two subproblems is immediate and the entire work happens in combining their solutions; here, the entire work happens in the division stage, with no work required to combine the solutions to the subproblems.

Here is pseudocode of quicksort: call *Quicksort(A[0..n − 1])*

ALGORITHM *Quicksort(A[l..r])*

//Sorts a subarray by quicksort

//Input: Subarray of array $A[0..n-1]$, defined by its left and right

//   indices l and r

//Output: Subarray $A[l..r]$ sorted in nondecreasing order

if $l < r$

    $s \leftarrow Partition(A[l..r])$ //s is a split position

    $Quicksort(A[l..s - 1])$

 $Quicksort(A[s + 1..r])$

As before, we start by selecting a pivot—an element with respect to whose value we are going to divide the subarray. There are several different strategies for selecting a pivot; we will return to this issue when we analyze the algorithm's efficiency. For now, we use the simplest strategy of selecting the subarray's first element: $p = A[l]$.

We will now scan the subarray from both ends, comparing the subarray's elements to the pivot. The left-to-right scan, denoted below by index pointer i, starts with the second element. Since we want elements smaller than the pivot to be in the left part of the subarray, this scan skips over elements that are smaller than the pivot and stops upon encountering the first element greater than or equal to the pivot. The right-to-left scan, denoted below by index pointer j, starts with the last element of the subarray. Since we want elements larger than the pivot to be in the right part of the subarray, this scan skips over elements that are larger than the pivot and stops on encountering the first element smaller than or equal to the pivot.

After both scans stop, three situations may arise, depending on whether or not the scanning indices have crossed. If scanning indices i and j have not crossed, i.e., $i < j$, we simply exchange $A[i]$ and $A[j]$ and resume the scans by incrementing i and decrementing j, respectively:

p	all are $\leq p$	$\geq p$	. .	$\leq p$	all are $\geq p$
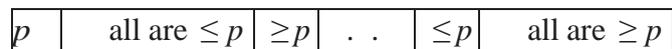
If the scanning indices have crossed over, i.e., $i > j$, we will have partitioned the subarray after exchanging the pivot with $A[j]$:

p	all are $\leq p$	$\leq p$	$\geq p$	all are $\geq p$
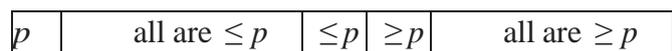
Finally, if the scanning indices stop while pointing to the same element, i.e., $i = j$, the value they are pointing to must be equal to p (why?). Thus, we have the subarray partitioned, with the split position $s = i = j$:

p	all are $\leq p$	$= p$	all are $\geq p$

We can combine the last case with the case of crossed-over indices $(i > j)$ by exchanging the pivot with $A[j]$ whenever $i \geq j$.

Here is pseudocode implementing this partitioning procedure.

ALGORITHM *Partition(A[l..r])*

//Partitions a subarray by Hoare's algorithm, using the first element as a pivot

//Input: Subarray of array $A[0..n-1]$, defined by its left and right indices l and r ($l < r$)

//Output: Partition of $A[l..r]$, with the split position returned as this function's value

$p \leftarrow A[l]; i \leftarrow l; j \leftarrow r + 1$

repeat

    repeat $i \leftarrow i + 1$ **until** $A[i] \geq p$

    repeat $j \leftarrow j - 1$ **until** $A[j] \leq p$

    swap(*A[i], A[j]*)

until $i \geq j$

swap(*A[i], A[j]*)   //undo last swap when $i \geq j$

swap(*A[l], A[j]*)

return *j*

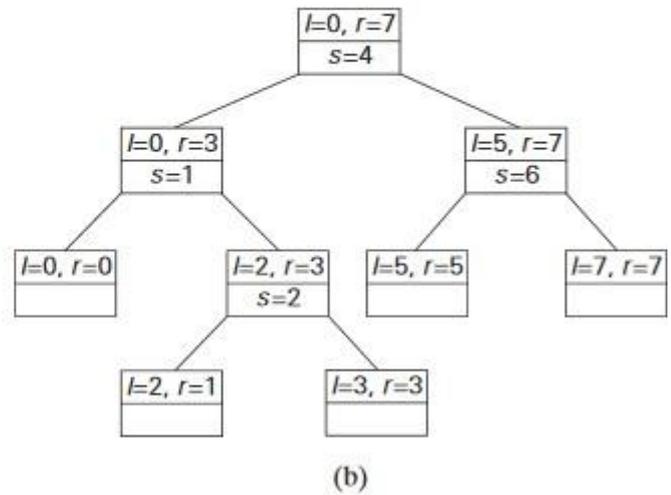. The number of key comparisons in the best case satisfies the recurrence

$$C_{best}(n) = 2C_{best}(n/2) + n \quad \text{for } n > 1, \ C_{best}(1) = 0.$$

According to the Master Theorem, $C_{best}(n) \in$ $(n \log_2 n)$; solving it exactly for

$n = 2^k$ yields $C_{best}(n) = n \log_2 n.$

0	1	2	3	4	5	6	7
5	3 (i)	1	9 (j)	8	2	4	7 (j)
5	3	1 (i)	9	8	2	4 (j)	7
5	3	1	4 (i)	8	2	9 (j)	7
5	3	1	4	8 (i)	2 (j)	9	7
5	3	1	4	2 (i)	8 (j)	9	7
5	3	1	4	2 (j)	8 (i)	9	7
2	3	1	4	5	8	9	7

2	3 (i)	1	4 (j)
2	3 (i)	1 (j)	4
2	1 (i)	3 (j)	4
2	1 (j)	3 (i)	4
1	2	3	4

3	4 (ij)
3	4 (i)
	4

8	9 (i)	7 (j)
8	7 (i)	9 (j)
8	7 (j)	9 (i)
7	8	9
7		

Tree (b):

- l=0, r=7 ; s=4
  - l=0, r=3 ; s=1
    - l=0, r=0
    - l=2, r=3 ; s=2
      - l=2, r=1
      - l=3, r=3
  - l=5, r=7 ; s=6
    - l=5, r=5
    - l=7, r=7

(b)

Dynamic programming:

Dynamic programming is a technique in computer science and mathematics that can be used to solve optimization problems by breaking them down into smaller subproblems and reusing solutions to those subproblems to solve the larger problem more efficiently.

The following are the key elements of dynamic programming:

1. **Overlapping subproblems:** A problem is said to have overlapping subproblems if the same subproblem is solved multiple times during the course of solving the larger problem. In dynamic programming, the solutions to these subproblems are stored in a table or cache, so thatthey can be reused later.

2. **Optimal substructure:** A problem is said to have optimal substructure if the optimal solution to the problem can be constructed from the optimal solutions to its subproblems. In other

words, if we know the optimal solutions to the subproblems, we can combine them to obtain the optimal solution to the larger problem.

3. **Memoization:** Memoization is a technique in dynamic programming where the solutions to subproblems are stored in a table or cache so that they can be reused later. This can significantly reduce the time complexity of the algorithm, especially for problems with overlapping subproblems.

4. **Recurrence relation:** A recurrence relation is a way of expressing a solution to a problem in terms of solutions to its subproblems. In dynamic programming, the recurrence relation is used to build the table or cache of solutions to the subproblems.

5. **Bottom-up approach:** In a bottom-up approach, we start by solving the subproblems and then combine the solutions to obtain the solution to the larger problem. This is in contrast to a top-down approach, where we start with the larger problem and recursively break it down into subproblems.

6. **State transition:** State transition is the process of moving from one state to another in the course of solving a dynamic programming problem. Each state represents a particular subproblem, and the state transition defines the relationship between the subproblems.

Optimal binary search tree

Optimal binary search tree (OBST) is a binary search tree that minimizes the expected cost of searching for a given set of keys. It is also known as a weight-balanced binary search tree or a dynamic binary search tree. OBST is a classic example of dynamic programming and can be solved using the dynamic programming approach.

The problem of constructing an OBST can be formulated as follows:

Given a sorted array of n keys (K1, K2, ..., Kn) with associated probabilities of search (p1, p2, ..., pn), construct an OBST that minimizes the expected cost of searching.

The expected cost of searching for a key Ki in the binary search tree can be defined as:

Cost(i, j) = Summation of (p[k]) from k=i to j + Cost(i, k-1) + Cost(k+1, j)

where i and j are the starting and ending indices of a subtree and k is the index of the root node of that subtree.
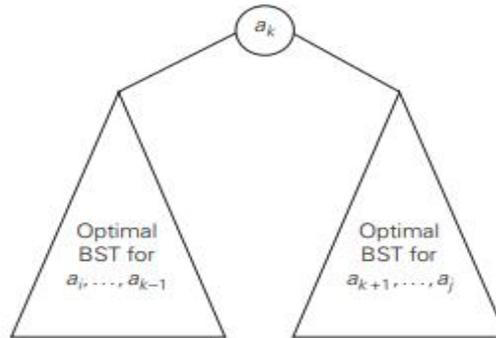
FIGURE 8.8 Binary search tree (BST) with root a_k and two optimal binary search subtrees T_i^{k-1} and T_{k+1}^j.

contains keys $a_{k+1}, \ldots, a_j$ also optimally arranged. (Note how we are taking advantage of the principle of optimality here.)

If we count tree levels starting with 1 to make the comparison numbers equal the keys' levels, the following recurrence relation is obtained:

$$C(i, j) = \min_{i \le k \le j} \{p_k \cdot 1 + \sum_{s=i}^{k-1} p_s \cdot (\text{level of } a_s \text{ in } T_i^{k-1} + 1)$$

$$+ \sum_{s=k+1}^{j} p_s \cdot (\text{level of } a_s \text{ in } T_{k+1}^j + 1)\}$$

$$= \min_{i \le k \le j} \{\sum_{s=i}^{k-1} p_s \cdot \text{level of } a_s \text{ in } T_i^{k-1} + \sum_{s=k+1}^{j} p_s \cdot \text{level of } a_s \text{ in } T_{k+1}^j + \sum_{s=i}^{j} p_s\}$$

$$= \min_{i \le k \le j} \{C(i, k-1) + C(k+1, j)\} + \sum_{s=i}^{j} p_s.$$

Thus, we have the recurrence

$$C(i, j) = \min_{i \le k \le j} \{C(i, k-1) + C(k+1, j)\} + \sum_{s=i}^{j} p_s \quad \text{for } 1 \le i \le j \le n. \quad \textbf{(8.8)}$$

We assume in formula (8.8) that $C(i, i - 1) = 0$ for $1 \le i \le n + 1$, which can be interpreted as the number of comparisons in the empty tree. Note that this formula implies that

$$C(i, i) = p_i \quad \text{for } 1 \le i \le n,$$

as it should be for a one-node binary search tree containing a_i.

FIGURE 8.9 Table of the dynamic programming algorithm for constructing an optimal binary search tree.

EXAMPLE   Let us illustrate the algorithm by applying it to the four-key set we used at the beginning of this section:

key	A	B	C	D
probability	0.1	0.2	0.4	0.3

The initial tables look like this:

main table

	0	1	2	3	4
1	0	0.1			
2		0	0.2		
3			0	0.4	
4				0	0.3
5					0

root table

|	0	1	2	3	4
1		1			
2			2		
3				3	
4					4
5					

Let us compute $C(1, 2)$:

$$C(1, 2) = \min \begin{cases} k = 1: & C(1, 0) + C(2, 2) + \sum_{s=1}^{2} p_s = 0 + 0.2 + 0.3 = 0.5 \\ k = 2: & C(1, 1) + C(3, 2) + \sum_{s=1}^{2} p_s = 0.1 + 0 + 0.3 = 0.4 \end{cases}$$

$$= 0.4.$$

Thus, out of two possible binary trees containing the first two keys, A and B, the root of the optimal tree has index 2 (i.e., it contains B), and the average number of comparisons in a successful search in this tree is 0.4.

We will ask you to finish the computations in the exercises. You should arrive at the following final tables:

main table

	0	1	2	3	4
1	0	0.1	0.4	1.1	1.7
2		0	0.2	0.8	1.4
3			0	0.4	1.0
4				0	0.3
5					0

root table

	0	1	2	3	4
1		1	2	3	3
2			2	3	3
3				3	3
4					4
5					
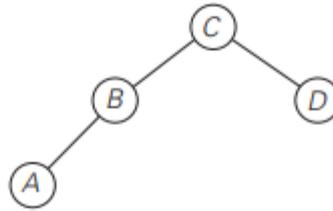
FIGURE 8.10 Optimal binary search tree for the example.

Here is pseudocode of the dynamic programming algorithm.

ALGORITHM *OptimalBST*$(P[1..n])$

//Finds an optimal binary search tree by dynamic programming
//Input: An array $P[1..n]$ of search probabilities for a sorted list of n keys
//Output: Average number of comparisons in successful searches in the
//   optimal BST and table R of subtrees' roots in the optimal BST
for $i \leftarrow 1$ **to** n **do**
　　$C[i, i-1] \leftarrow 0$
　　$C[i, i] \leftarrow P[i]$
　　$R[i, i] \leftarrow i$
$C[n+1, n] \leftarrow 0$
for $d \leftarrow 1$ **to** $n-1$ **do**　//diagonal count
　　for $i \leftarrow 1$ **to** $n-d$ **do**
　　　　$j \leftarrow i + d$
　　　　$minval \leftarrow \infty$
　　　　for $k \leftarrow i$ **to** j **do**
　　　　　　if $C[i, k-1] + C[k+1, j] < minval$
　　　　　　　　$minval \leftarrow C[i, k-1] + C[k+1, j];$　$kmin \leftarrow k$
　　　　$R[i, j] \leftarrow kmin$
　　　　$sum \leftarrow P[i];$　**for** $s \leftarrow i+1$ **to** j **do** $sum \leftarrow sum + P[s]$
　　　　$C[i, j] \leftarrow minval + sum$
return $C[1, n], R$

Greedy Technique:

Elements of the greedy strategy

In computer science and optimization, a greedy strategy is a method that makes locally optimal choices at each step with the hope of finding a global optimum. The strategy is based on the idea that making the best decision at each step will lead to an overall optimal solution. However, the strategy may not always work and can sometimes lead to suboptimal solutions.

Here are some elements of a greedy strategy:

1. **Greedy choice property:** A greedy strategy works by making locally optimal choices at each step. This means that at each step, we choose the best possible option without considering the future consequences. The choice that appears to be the best at the moment is made, without worrying about whether this choice will lead to the best overall solution.

2. **Optimal substructure property:** A problem exhibits optimal substructure if an optimal solution to the problem contains optimal solutions to its subproblems. This means that we can break down the problem into smaller subproblems, solve each subproblem optimally, and then combine the solutions to get the overall optimal solution.

3. **Greedy algorithm:** A greedy algorithm is a method for finding an optimal solution that uses a greedy strategy. The algorithm starts with an empty solution and iteratively adds elements to the solution, always choosing the element that appears to be the best at the moment. Once all the elements have been added, the solution is complete.

4. **Proof of correctness:** To prove that a greedy algorithm is correct, we need to show that it always produces an optimal solution. This can be done using either a direct proof or an exchange argument, where we show that any suboptimal solution can be transformed into an optimal solution without violating the greedy choice property.

Examples of problems that can be solved using a greedy strategy: Some examples of problems that can be solved using a greedy strategy include the coin change problem, the activity selection problem, and the Huffman coding problem.

HUFFMAN TREES

Huffman trees are constructed for encoding a given text of characters, each character is associated (converted) with some bit sequence. This bit sequence is named as *code word*.

The encoding is classified into two types, depends on the number of bits used for each character in the text.

(a) *Fixed length encoding* - each character is associated with a bit string of some fixed length.

Example : ASCII - 7 bits for a character.

(b) *Variable length encoding* - each character is associated with a bit string of different length. (i.e.)

(i) Shorter length codeword for more frequent characters and longer length codeword for less frequent characters. Example Telegraph code.

(ii) Using the property of prefix code - no codeword is a prefix of another character's code word. This property is used to identify the number of bits required to encode the ith character of a text.

How to generate binary prefix code? Associate the character of a text to be encoded with leaves of a binary tree, in which all the left edges are labelled by 0 and all the right edges are labelled by 1.

How to assign shorter bit strings to high frequency characters and longer bit strings to low frequency character strings? This task is done by the greedy algorithm given below.

Huffman's Algorithm

Step 1 Initialize n one-node trees and label them with the characters of the alphabet. Record the frequency of each character in its tree's root to indicate the tree's weight. (More generally, the weight of a tree will be equal to the sum of the frequencies in the tree's leaves)

Step 2 Repeat the following operation until a single tree is obtained. Find two trees with the smallest weight (ties can be broken arbitrarily). Make them the left and right subtree of a new tree and record the sum of their weights in the root of the new tree as its weight.

A tree constructed by this algorithm is called a *Huffman tree*. It defines the character form of strings (code), based on their frequencies in a given text is known as *Huffman*

A tree constructed by the above algorithm is called a **Huffman tree**. It defines—in the manner described above—a **Huffman code**.

EXAMPLE   Consider the five-symbol alphabet {A, B, C, D, _} with the following occurrence frequencies in a text made up of these symbols:

symbol	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15
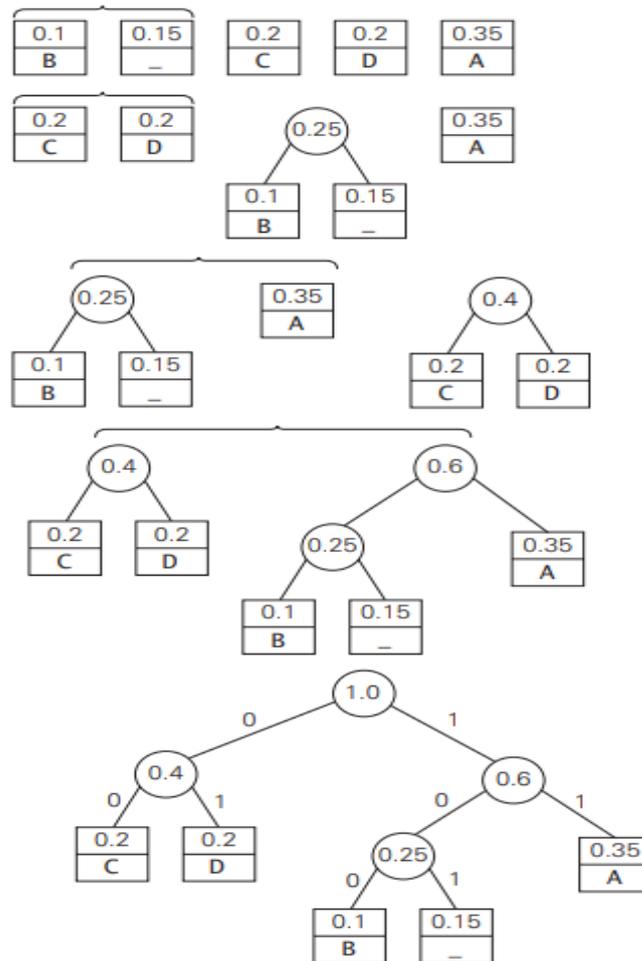
FIGURE 9.12 Example of constructing a Huffman coding tree.

The resulting codewords are as follows:

symbol	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15
codeword	11	100	00	01	101

Hence, DAD is encoded as 011101, and 10011011011101 is decoded as BAD_AD.

3.6 Multi Stage Graph

Problem Description : A multistage graph G = (V, E) which is a directed graph. In this graph all the vertices are partitioned into the k stages where k ≥ 2. In multistage graph problem we have to find the shortest path from source to sink. The cost of each path is calculated by using the weight given along that edge. The cost of a path from source (denoted by S) to sink (denoted by T) is the sum of the costs of edges on the path. In multistage graph problem we have to find the path from S to T. There is set of vertices in each stage. The multistage graph can be solved using forward and backward approach.

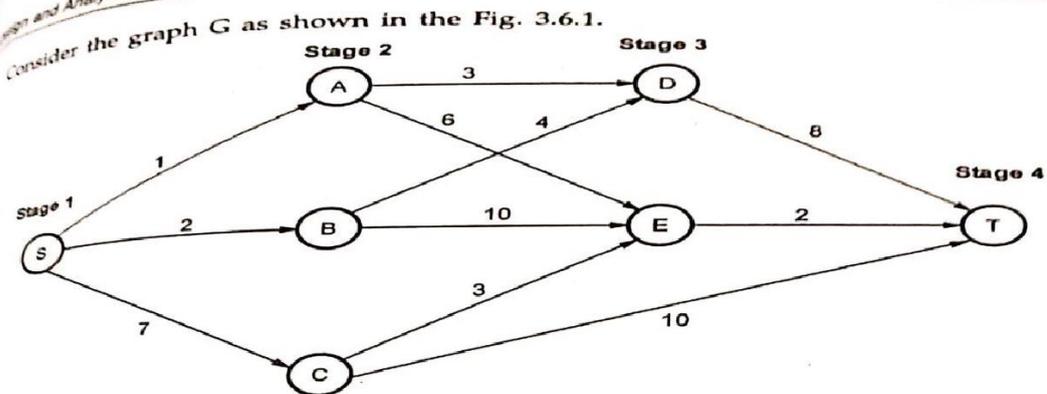Let us solve multistage problem for both the approaches with the help of some example.

Consider the graph G as shown in the Fig. 3.6.1.

Fig. 3.6.1 Multistage graph

There is single vertex in stage 1, then 3 vertices in stage 2, then 2 vertices in stage 3 and only one vertex in stage 4 (this is a target stage).

Backward approach

$$d(S, T) = \min \{1 + d(A, T), 2 + d(B, T), 7 + d(C, T)\} \qquad \dots (1)$$

We will now compute d(A, T), d(B, T) and d(C,T).

$$d(A, T) = \min \{3 + d(D, T), 6 + d(E, T)\} \qquad \dots (2)$$

$$d(B, T) = \min \{4 + d(D, T), 10 + d(E, T)\} \qquad \dots (3)$$

$$d(C, T) = \min \{3 + d(E, T), d(C, T)\} \qquad \dots (4)$$

Now let us compute d(D, T) and d(E, T).

$$d(D, T) = 8$$

$$d(E, T) = 2 \qquad \qquad \textbf{backward vertex = E}$$

Let us put these values in equations (2), (3) and (4).

$$d(A, T) = \min \{3 + 8, 6 + 2\}$$

$$d(A, T) = 8 \qquad \qquad \textbf{A - E - T}$$

$$d(B, T) = \min \{4 + 8, 10 + 2\}$$

$$d(B, T) = 12 \qquad \qquad \textbf{A - D - T}$$

$$d(C, T) = \min \{3 + 2, 10\}$$

$$d(C, T) = 5 \qquad \qquad \textbf{C - E - T}$$

$$\therefore \quad d(S, T) = \min\{1 + d(A, T), 2 + d(B, T), 7 + d(C, T)\}$$
$$= \min\{1 + 8, 2 + 12, 7 + 5\}$$
$$= \min\{9, 14, 12\}$$
$$d(S, T) = 9 \qquad\qquad S - A - E - T$$

Optimal Merge Pattern
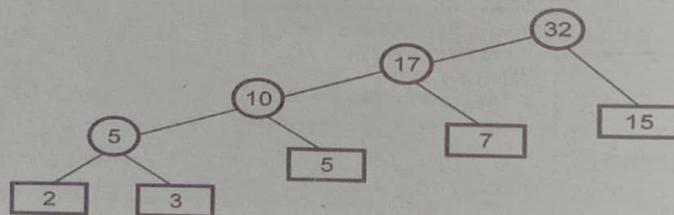
3.14 Optimal Merge Pattern

- The merge sort is a technique in which two sorted files can be merged together in one sorted file. When two or more files need to be merged in a single file, then the merging of two sorted files must be performed repeatedly. That means merging of sorted files must be done pair-wise.
- For instance : If there are a, b, c and d files which are already sorted and has to be merged then -
  1. We may merge a, b to produce x_1 then x_1 can be merged with file c to produce x_2, then x_2 can be merged with file d to produce x_3.
  2. We may merge a, b to produce x_1, then merge c and d to produce x_2 finally merge x_1 and x_2 to produce x_3.
- Thus different pairings can be possible to produce final merged file. Clearly, these pairing may require different amount of computing time.
- The pairing of files which require optimum (minimum) computing time to get a single merged file denotes the **optimal merge pattern.**
- For example :

Consider that there are 3 files namely a, b, c of lengths 11, 12 and 13 respectively. Then following are two sample merge patterns that can be produced.

Pattern 2 is faster than first pattern. A Greedy method is used to obtain optimal merge pattern.

Representation of optimal merge pattern

The merge pattern is typically represented by a binary tree, it is also called as **two way merge pattern.** For example 2, 3, 5, 7, 15 are six files with given lengths then we can build a merge pattern as

The leaf nodes denoted by squares are called **external nodes** and remaining nodes are called **internal nodes.** The path length from root to corresponding external node is called **external path length.**

An application of optimal merge pattern is **Huffman code.**

Activity Selection Problem:

The Activity Selection Problem is an optimization problem which deals with the selection of non-conflicting activities that needs to be executed by a single person or machine in a given time frame. Each activity is marked by a start and finish time. Greedy technique is used for finding the solution since this is an optimization problem.

What is Activity Selection Problem?
Let's consider that you have n activities with their start and finish times, the objective is to find solution set having **maximum number of non-conflicting activities** that can be executed in a single time frame, assuming that only one person or machine is available for execution.

- It might not be possible to complete all the activities, since their timings can collapse.
- Two activities, say **i** and **j**, are said to be non-conflicting if $s_i \geq f_j$ or $s_j \geq f_i$ where s_i and s_j denote the starting time of activities **i** and **j** respectively, and f_i and f_j refer to the finishing time of the activities **i** and **j** respectively.
- **Greedy approach** can be used to find the solution since we want to maximize the count of activities that can be executed. This approach will greedily choose an activity with earliest finish time at every step, thus yielding an optimal solution.

Input Data for the Algorithm:

- act[] array containing all the activities.
- s[] array containing the starting time of all the activities.
- f[] array containing the finishing time of all the activities.

Ouput Data from the Algorithm:

- sol[] array refering to the solution set containing the maximum number of non-conflicting activities.

Steps for Activity Selection Problem

Following are the steps we will be following to solve the activity selection problem,

Step 1: Sort the given activities in ascending order according to their finishing time.

Step 2: Select the first activity from sorted array act[] and add it to sol[] array.

Step 3: Repeat steps 4 and 5 for the remaining activities in act[].

Step 4: If the start time of the currently selected activity is greater than or equal to the finish time of previously selected activity, then add it to the sol[] array.

Step 5: Select the next activity in act[] array.

Step 6: Print the sol[] array.

Activity Selection Problem Example

Let's try to trace the steps of above algorithm using an example:

In the table below, we have 6 activities with corresponding start and end time, the objective is to compute an execution schedule having maximum number of non-conflicting activities:

Start Time (s)	Finish Time (f)	Activity Name
5	9	a1
1	2	a2
3	4	a3
0	6	a4
5	7	a5
8	9	a6

A possible **solution** would be:

Step 1: Sort the given activities in ascending order according to their finishing time.

The table after we have sorted it:

Start Time (s)	Finish Time (f)	Activity Name
1	2	a2
3	4	a3
0	6	a4
5	7	a5
5	9	a1
8	9	a6

Step 2: Select the first activity from sorted array act[] and add it to the sol[] array, thus **sol = {a2}.Step 3**:

Repeat the steps 4 and 5 for the remaining activities in act[].

Step 4: If the start time of the currently selected activity is greater than or equal to the finish time ofthe previously selected activity, then add it to sol[].

Step 5: Select the next activity in act[]For the

data given in the above table,

A. Select activity **a3**. Since the start time of **a3** is greater than the finish time of **a2** (i.e. s(a3) >f(a2)), we add **a3** to the solution set. Thus **sol = {a2, a3}**.

B. Select **a4**. Since s(a4) < f(a3), it is not added to the solution set.

C. Select **a5**. Since s(a5) > f(a3), **a5** gets added to solution set. Thus **sol = {a2, a3, a5}**

D. Select **a1**. Since s(a1) < f(a5), **a1** is not added to the solution set.

E. Select **a6**. **a6** is added to the solution set since s(a6) > f(a5). Thus **sol = {a2, a3, a5, a6}**.**Step 6**: At

last, print the array sol[]

Hence, the execution schedule of maximum number of non-conflicting activities will be:
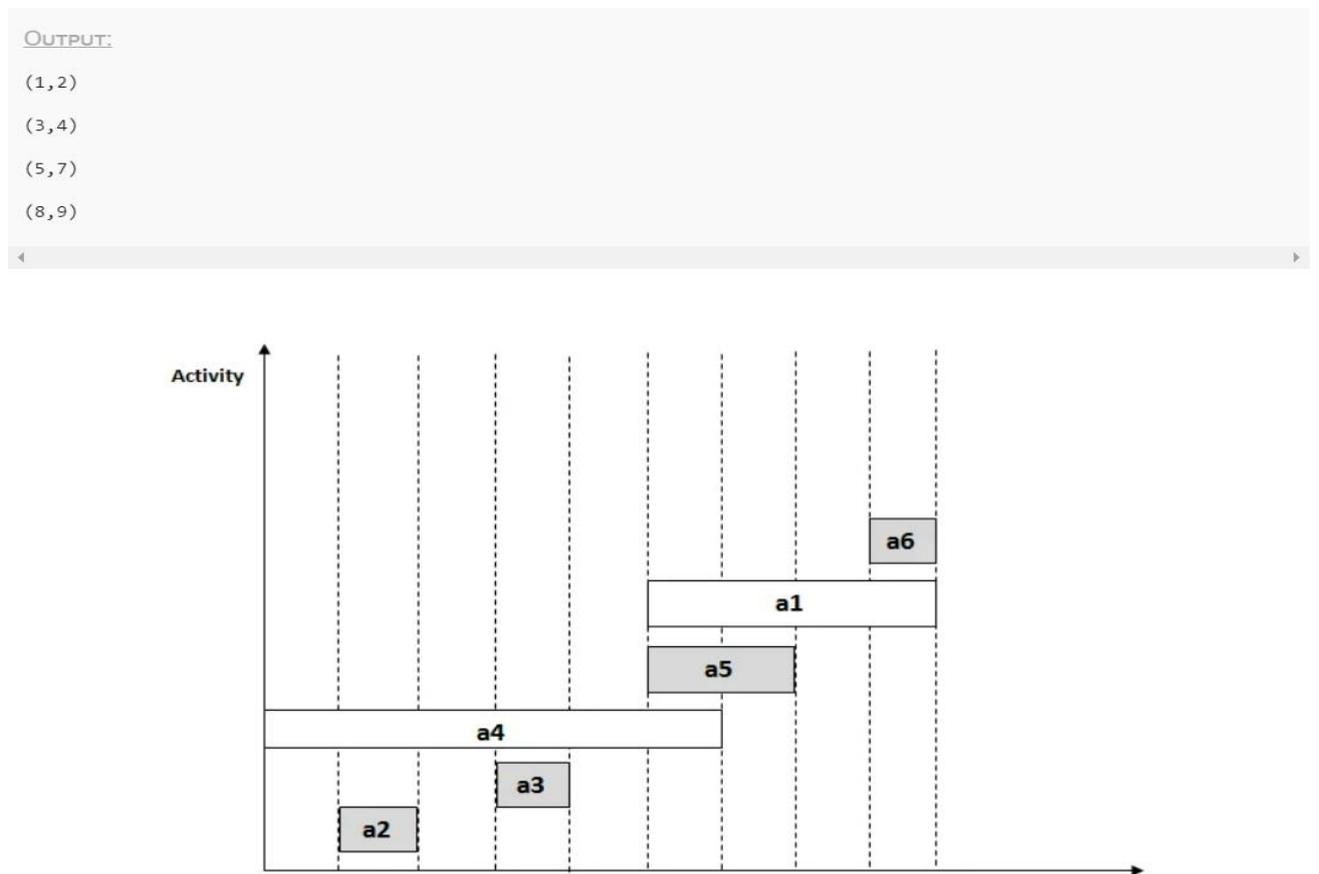
OUTPUT:
(1,2)
(3,4)
(5,7)
(8,9)

Backtracking: n-Queens problem - Hamiltonian Circuit Problem - Subset Sum Problem – Graph colouring problem **Branch and Bound**: Solving 15-Puzzle problem - Assignment problem - Knapsack Problem - Travelling Salesman Problem

5.6 BACKTRACKING

- Backtracking is a more intelligent variation approach.
- The principal idea is to construct solutions one component at a time and evaluate such partially constructed candidates as follows.
- If a partially constructed solution can be developed further without violating the problem's constraints, it is done by taking the first remaining legitimate option for the next component.
- If there is no legitimate option for the next component, no alternatives for any remaining component need to be considered. In this case, the algorithm backtracks to replace the last component of the partially constructed solution with its next option.
- It is convenient to implement this kind of processing by constructing a tree of choices being made, called **the state-space tree**.
- Its root represents an initial state before the search for a solution begins.
- The nodes of the first level in the tree represent the choices made for the first component of a solution, the nodes of the second level represent the choices for the second component, and so on.
- A node in a state-space tree is said to be promising if it corresponds to a partially constructed solution that may still lead to a complete solution. otherwise, it is called *nonpromising*.
- Leaves represent either nonpromising dead ends or complete solutions found by the algorithm. In the majority of cases, a statespace tree for a backtracking algorithm is constructed in the manner of depthfirst search.
- If the current node is promising, its child is generated by adding the first remaining legitimate option for the next component of a solution, and the processing moves to this child. If the current node turns out to be nonpromising, the algorithm backtracks to the node's parent to consider the next possible option for its last component; if there is no such option, it backtracks one more level up the tree, and so on.
- Finally, if the algorithm reaches a complete solution to the problem, it either stops (if just one solution is required) or continues searching for other possible solutions.
- Backtracking techniques are applied to solve the following problems
    - *n*-Queens Problem
    - Hamiltonian Circuit Problem
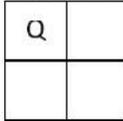   - Subset-Sum Problem

5.7 N-QUEENS PROBLEM

The problem is to place n queens on an $n \times n$ chessboard so that no two queens attack each other by being in the same row or in the same column or on the same diagonal.
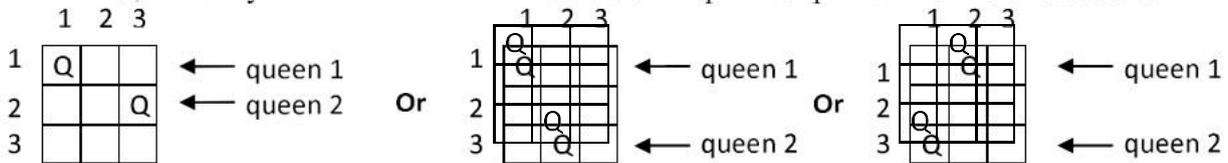
For $n = 1$, the problem has **a trivial solution**.

Q

For n = 2, it is easy to see that there is **no solution** to place 2 queens in 2×2 chessboard.
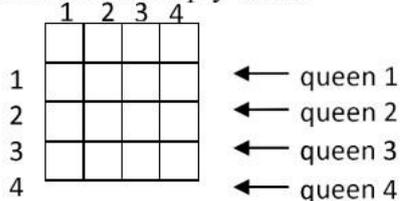
Q	

For $n = 3$, it is easy to see that there is **no solution** to place 3 queens in 3×3 chessboard.
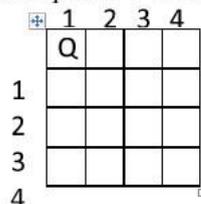

For $n = 4$, There is **solution** to place 4 queens in 4×4 chessboard. the four-queens problem solved by the backtracking technique.

Step 1: Start with the empty board

Step 2: Place queen 1 in the first possible position of its row, which is in column 1 of row 1.

Step 3: place queen 2, after trying unsuccessfully columns 1 and 2, in the first acceptable position for it, which is square (2, 3), the square in row 2 and column 3.
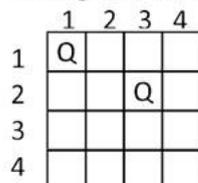
Step 4: This proves to be a dead end because there is no acceptable position for queen 3. So, the algorithm backtracks and puts queen 2 in the next possible position at (2, 4).

	1	2	3	4
1	Q			
2				Q
3				
4				

Step 5: Then queen 3 is placed at (3, 2), which proves to be another dead end.

	1	2	3	4
1	Q			
2				Q
3		Q		
4				

Step 6: The algorithm then backtracks all the way to queen 1 and moves it to (1, 2).

	1	2	3	4
1		Q		
2				
3				
4				

Step 7: The queen 2 goes to (2, 4).

	1	2	3	4
1		Q		
2				Q
3				
4				

Step 8: The queen 3 goes to (3, 1).

	1	2	3	4
1		Q		
2				Q
3	Q			
4				

Step 9: The queen 3 goes to (4, 3). This is a solution to the problem.

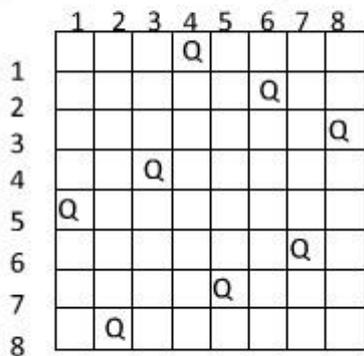|	1	2	3	4
1		Q		
2				Q
3	Q			
4			Q	

FIGURE 5.9 Solution four-queens problem in 4x4 Board.

The state-space tree of this search is shown in Figure 12.2
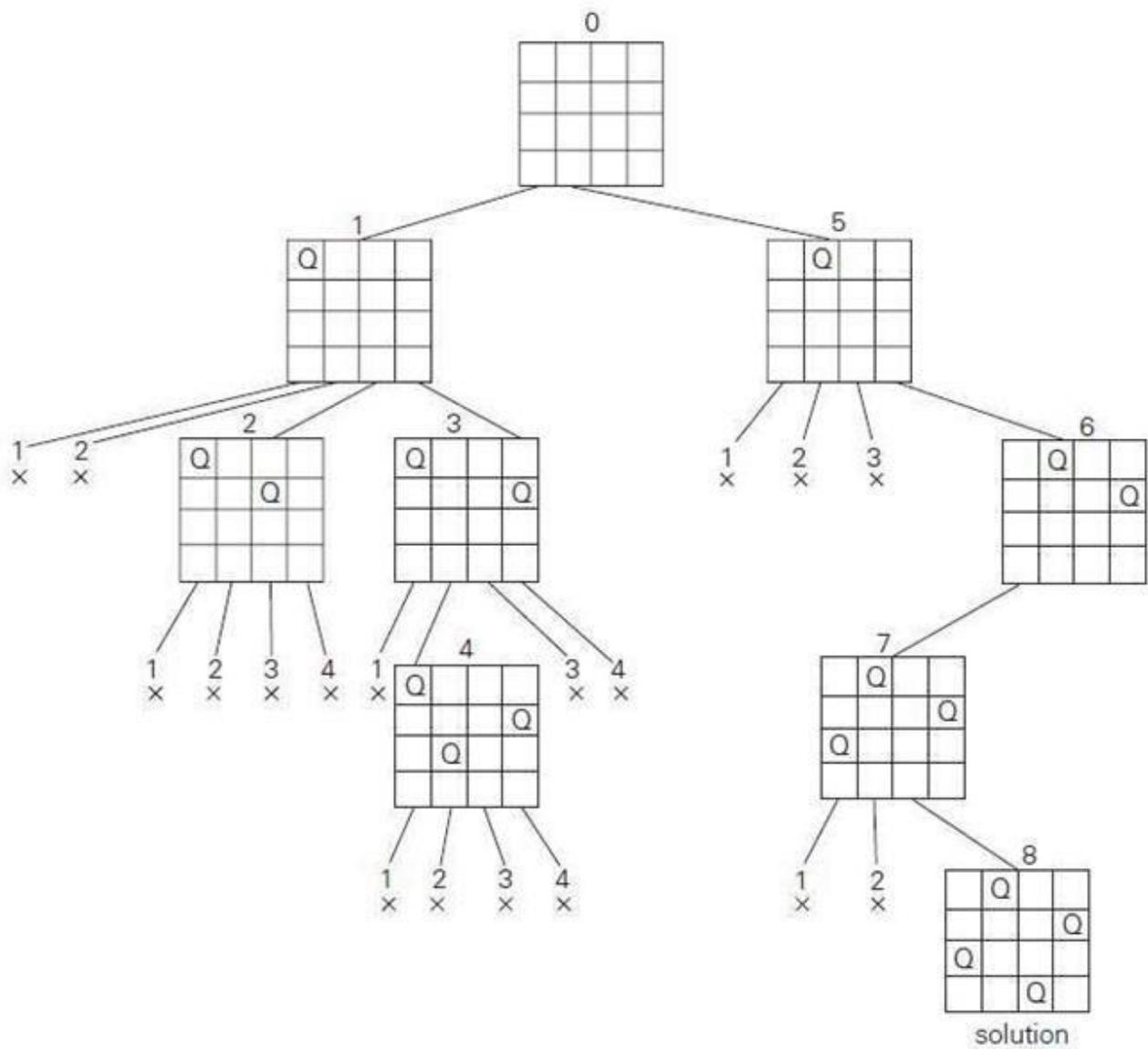
FIGURE 5.10 State-space tree of solving the four-queens problem by backtracking. × denotes an unsuccessful attempt to place a queen.

For n = 8. There is **solution** to place 8 queens in 8 × 8 chessboard.

FIGURE 5.11 Solution 8-queens problem in 8x8 Board.
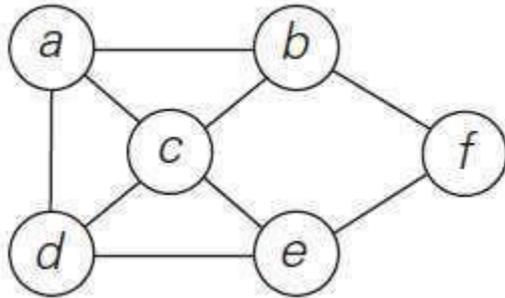
5.8 HAMILTONIAN CIRCUIT PROBLEM

A **Hamiltonian circuit** (also called a **Hamiltonian cycle**, **Hamilton cycle**, or **Hamilton circuit**) is a graph **cycle** (i.e., closed loop) through a graph that visits each node exactly once. A graph possessing a **Hamiltonian cycle** is said to be a **Hamiltonian** graph.
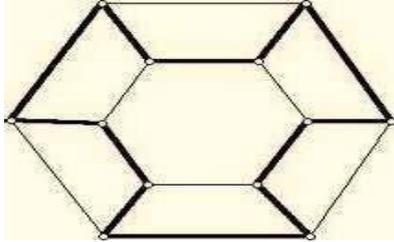
FIGURE 5.12 Graph contains Hamiltonian circuit

Let us consider the problem of finding a Hamiltonian circuit in the graph in Figure 5.13.

Example: Find Hamiltonian circuit starts at vertex a.

FIGURE 5. 13 Graph.

Solution:
- Assume that if a Hamiltonian circuit exists, it starts at vertex a. accordingly, we make vertex a the root of the state-space tree as in Figure 5.14.
- In a Graph G, Hamiltonian cycle begins at some vertex $V_1 \in$ G, and the vertices are visited only once in the order $V_1, V_2, \ldots, V_n.$ (V_i are distinct except for V_1 and V_{n+1} which are equal).
- The first component of our future solution, if it exists, is a first intermediate vertex of a Hamiltonian circuit to be constructed. Using the alphabet order to break the three-way tie among the vertices adjacent to a, we
- Select vertex b. From b. the algorithm proceeds to c. then to d. then to e. and finally to f, which proves to be a dead end.
- So the algorithm backtracks from f to e. then to d. and then to c. which provides the first alternative for the algorithm to pursue.
- Going from c to e eventually proves useless, and the algorithm has to backtrack from e to c and then to b. From there, it goes to the vertices f, e, c, and d, from which it can legitimately return to a. yielding the Hamiltonian circuit a, b, f, e, c, d, a. If we wanted to find another Hamiltonian circuit, we could continue this process by backtracking from the leaf of the solution found.
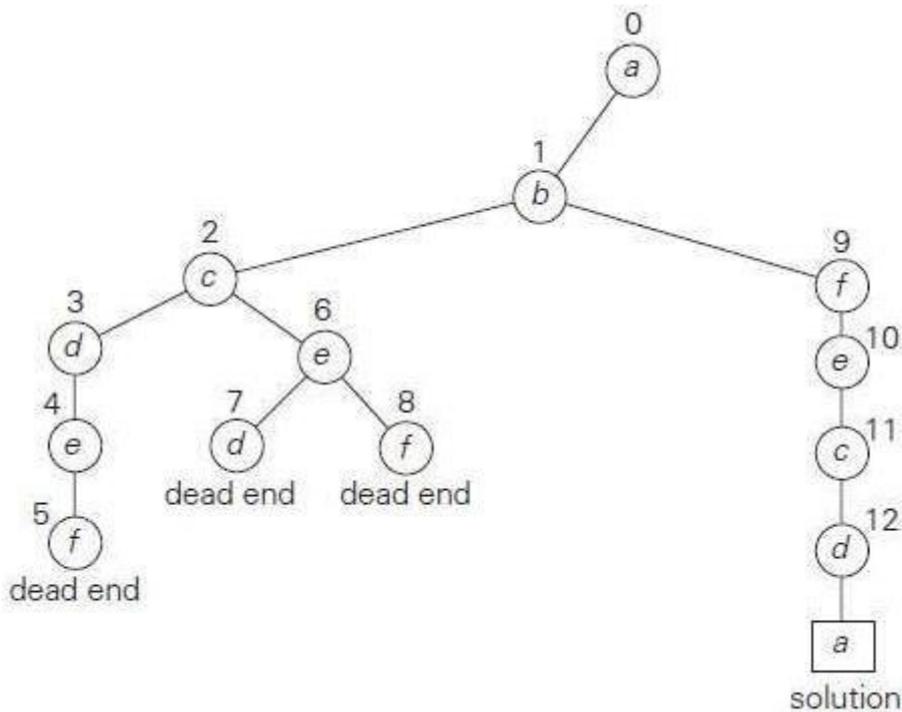
FIGURE 5.14 State-space tree for finding a Hamiltonian circuit.

5.9 SUBSET SUM PROBLEM

The ***subset-sum problem*** finds a subset of a given set $A = \{a1, \ldots, an\}$ of n positive integers whose sum is equal to a given positive integer d. For example, for $A = \{1, 2, 5, 6, 8\}$ and $d = 9$, there are two solutions: $\{1, 2, 6\}$ and $\{1, 8\}$. Of course, some instances of this problem may have no solutions.

It is convenient to sort the set's elements in increasing order. So, we will assume that

$$a1 < a2 < \ldots < an.$$

$A = \{3, 5, 6, 7\}$ and $d = 15$ of the subset-sum problem. The number inside a node is the sum of the elements already included in the subsets represented by the node. The inequality below a leaf
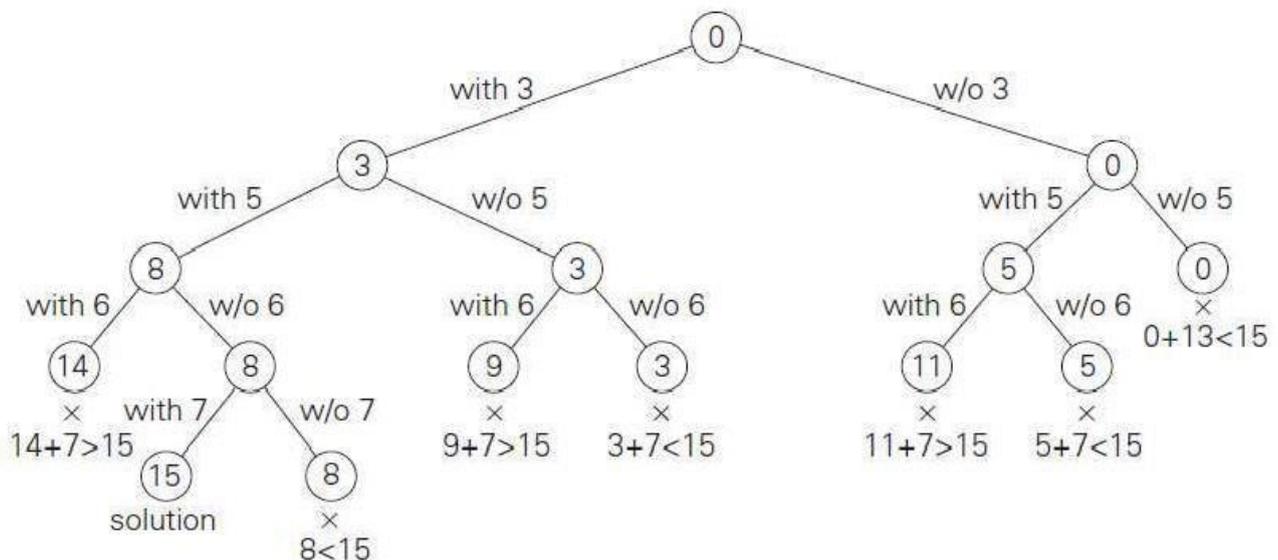

FIGURE 5.15 Complete state-space tree of the backtracking algorithm applied to the instance

Example:

- The state-space tree can be constructed as a binary tree like that in Figure 5.15 for the instance $A = \{3, 5, 6, 7\}$ and $d = 15$.
- The root of the tree represents the starting point, with no decisions about the given elements made as yet.
- Its left and right children represent, respectively, inclusion and exclusion of a_1 in a set being sought. Similarly, going to the left from a node of the first level corresponds to inclusion of a_2 while going to the right corresponds to its exclusion, and so on.
- Thus, a path from the root to a node on the ith level of the tree indicates which of the first I numbers have been included in the subsets represented by that node.
- We record the value of s, the sum of these numbers, in the node.
- If s is equal to d, we have a solution to the problem. We can either report this result and stop or, if all the solutions need to be found, continue by backtracking to the node's parent.
- If s is not equal to d, we can terminate the node as nonpromising if either of the following two inequalities holds:

$$s + a_{i+1} > d \text{ (the sum } s \text{ is too large)},$$
$$s + \sum_{j=i+1}^{n} a_j < d \text{ (the sum } s \text{ is too small)}.$$

General Remarks

From a more general perspective, most backtracking algorithms fit the following escription. An output of a backtracking algorithm can be thought of as an n-tuple $(x_1, x_2, \ldots, x_n)$ where each coordinate xi is an element of some finite lin early ordered set Si . For example, for the n-queens problem, each Si is the set of integers (column numbers) 1 through n.

A backtracking algorithm generates, explicitly or implicitly, a state-space tree; its nodes represent partially constructed tuples with the first i coordinates defined by the earlier actions of the algorithm. If such a tuple $(x_1, x_2, \ldots, x_i)$ is not a solution, the algorithm finds the next element in S_{i+1} that is consistent with the values of $((x_1, x_2, \ldots, x_i)$ and the problem's constraints, and adds it to the tuple as its (i + 1)st coordinate. If such an element does not exist, the algorithm backtracks to consider the next value of xi, and so on.

> **ALGORITHM** *Backtrack(X [1..i])*
>   //Gives a template of a generic backtracking algorithm
>   //Input: $X[1..i]$ specifies first i promising components of a solution
>   //Output: All the tuples representing the problem's solutions
>   **if** $X[1..i]$ is a solution **write** $X[1..i]$
>   **else**   //see Problem this section
>
>   **for** each element $x \in Si+1$ consistent with $X[1..i]$ and the constraints **do**
> $X[i + 1] \leftarrow x$
>   *Backtrack(X[1..i + 1])*

5.10 BRANCH AND BOUND

An optimization problem seeks to minimize or maximize some objective function, usually subject to some constraints. Note that in the standard terminology of optimization problems, a *feasible solution* is a point in the problem's search space that satisfies all the problem's constraints (e.g., a Hamiltonian circuit in the travelling salesman problem or a subset of items whose total weight does not exceed the knapsack's capacity in the knapsack problem), whereas an *optimal solution* is a feasible solution with the best value of the objective function (e.g., the shortest Hamiltonian circuit or the most valuable subset of items that fit the knapsack).

Compared to backtracking, branch-and-bound requires two additional items:
1. a way to provide, for every node of a state-space tree, a bound on the best value of the objective function1 on any solution that can be obtained by adding further components to the partially constructed solution represented by the node
2. the value of the best solution seen so far

If this information is available, we can compare a node's bound value with the value of the best solution seen so far. If the bound value is not better than the value of the best solution seen so

far—i.e., not smaller for a minimization problem and not larger for a maximization problem—the node is nonpromising and can be terminated (some people say the branch is "pruned"). Indeed, no solution obtained from it can yield a better solution than the one already available. This is the principal idea of the branch-and-bound technique.

In general, we terminate a search path at the current node in a state-space tree of a branch-and-bound algorithm for any one of the following three reasons:
1. The value of the node's bound is not better than the value of the best solution seen so far.

2. The node represents no feasible solutions because the constraints of the problem are already violated.

3. The subset of feasible solutions represented by the node consists of a single point (and hence no further choices can be made)—in this case, we compare the value of the objective function for this feasible solution with that of the best solution seen so far and update the latter with the former if the new solution is better.

Some problems can be solved by Branch-and-Bound are:
1. Assignment Problem
2. Knapsack Problem
3. Traveling Salesman Problem
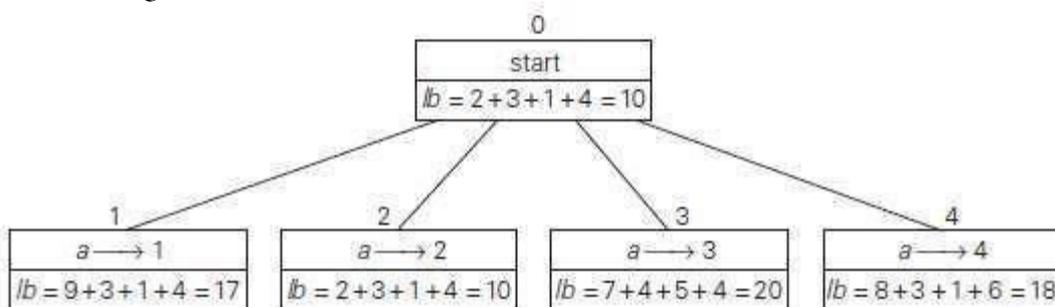
5.11 ASSIGNMENT PROBLEM

Let us illustrate the branch-and-bound approach by applying it to the problem of assigning n people to n jobs so that the total cost of the assignment is as small as possible. An instance of the assignment problem is specified by an $n \times n$ cost matrix C.

$$
C = \begin{bmatrix}
9 & 2 & 7 & 8 \\
6 & 4 & 3 & 7 \\
5 & 8 & 1 & 8 \\
7 & 6 & 9 & 4
\end{bmatrix}
\begin{matrix}
\text{person } a \\
\text{person } b \\
\text{person } c \\
\text{person } d
\end{matrix}
$$

with columns labeled job 1, job 2, job 3, job 4

We have to find a lower bound on the cost of an optimal selection without actually solving the problem. We can do this by several methods. For example, it is clear that the cost of any solution, including an optimal one, cannot be smaller than the sum of the smallest elements in each of the matrix's rows. For the instance here, this sum is $2 + 3 + 1 + 4 = 10$. It is important to stress that this is not the cost of any legitimate selection (3 and 1 came from the same column of the matrix); it is just a lower bound on the cost of any legitimate selection. We can and will apply the same thinking to partially constructed solutions. For example, for any legitimate selection that selects 9 from the first row, the lower bound will be $9 + 3 + 1 + 4 = 17$.

It is sensible to consider a node with the best bound as most promising, although this does not, of course, preclude the possibility that an optimal solution will ultimately belong to a different branch of the state-space tree. This variation of the strategy is called the ***best-first branch-and-bound***.

The lower-bound value for the root, denoted lb, is 10. The nodes on the first level of the tree correspond to selections of an element in the first row of the matrix, i.e., a job for person a *as shown in* Figure 5.15.

FIGURE 5.15 Levels 0 and 1 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm. The number above a node shows the order in which the node was generated. A node's fields indicate the job number assigned to person a and the lower bound value, lb, for this node.

So we have four live leaves (promising leaves are also called ***live***) —nodes 1 through 4— that may contain an optimal solution. The most promising of them is node 2 because it has the smallest lowerbound value. Following our best-first search strategy, we branch out from that node first by considering the three different ways of selecting an element from the second row and not in the second column—the three different jobs that can be assigned to person b (Figure 5.16).
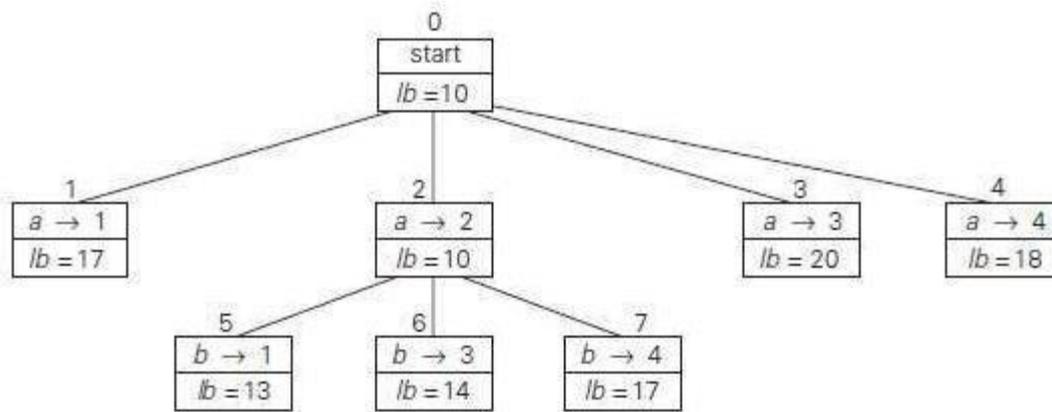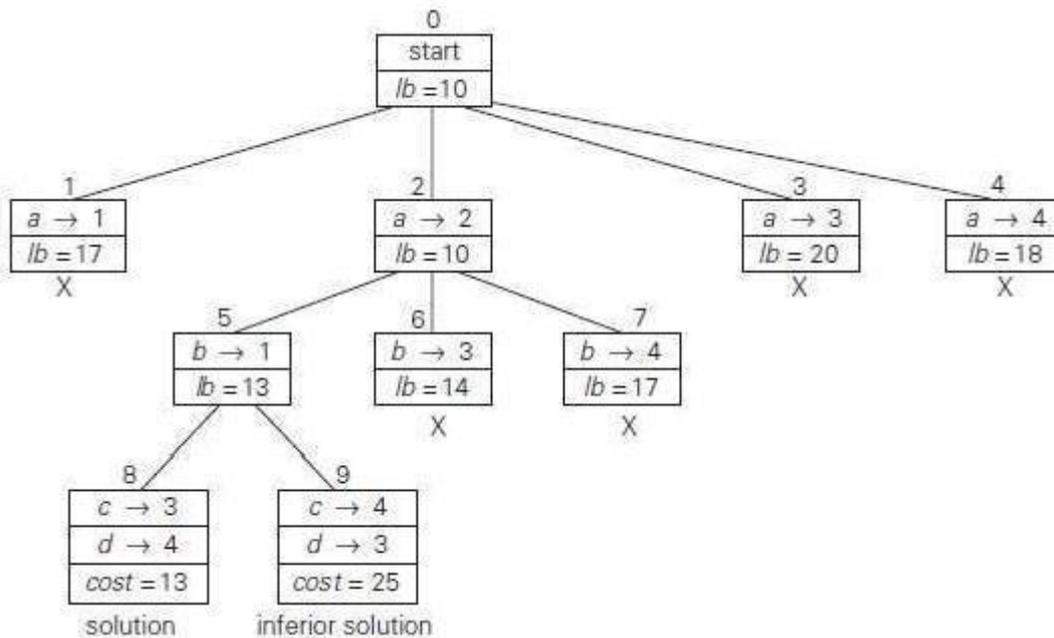
FIGURE 5.16 Levels 0, 1, and 2 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm.

Of the six live leaves—nodes 1, 3, 4, 5, 6, and 7—that may contain an optimal solution, we again choose the one with the smallest lower bound, node 5. First, we consider selecting the third column's element from c's row (i.e., assigning person c to job 3); this leaves us with no choice but to select the element from the fourth column of d's row (assigning person d to job 4). This yields leaf 8 (Figure 5.17), which corresponds to the feasible solution $\{a{\rightarrow}2, b{\rightarrow}1, c{\rightarrow}3, d{\rightarrow}4\}$ with the total cost of 13. Its sibling, node 9, corresponds to the feasible solution $\{a{\rightarrow}2, b{\rightarrow}1, c{\rightarrow}4, d{\rightarrow}3\}$ with the total cost of 25. Since its cost is larger than the cost of the solution represented by leaf 8, node 9 is simply terminated. (Of course, if its cost were smaller than 13, we would have to replace the information about the best solution seen so far with the data provided by this node.)

FIGURE 5.17 Complete state-space tree for the instance of the assignment problem solved with the best-first branch-and-bound algorithm.

Now, as we inspect each of the live leaves of the last state-space tree—nodes 1, 3, 4, 6, and 7 in Figure 5.17—we discover that their lower-bound values are not smaller than 13, the value of the best selection seen so far (leaf 8). Hence, we terminate all of them and recognize the solution represented by leaf 8 as the optimal solution to the problem.

5.12 KNAPSACK PROBLEM

discuss how we can apply the branch-and-bound technique to solving the knapsack problem. Given n items of known weights w_i and values v_i, $i = 1, 2, \ldots, n$, and a knapsack of capacity W, find the most valuable subset of the items that fit in the knapsack. It is convenient to order the items of a given instance in descending order by their value-to-weight ratios. Then the first item gives the best payoff per weight unit and the last one gives the worst payoff per weight unit, with ties resolved arbitrarily:

$$v1/w1 \geq v2/w2 \geq \ldots \geq vn/wn.$$

It is natural to structure the state-space tree for this problem as a binary tree constructed as follows. Each node on the ith level of this tree, $0 \leq i \leq n$, represents all the subsets of n items that include a particular selection made from the first i ordered items. This particular selection is uniquely determined by the path from the root to the node: a branch going to the left indicates the inclusion of the next item, and a branch going to the right indicates its exclusion. We record the total weight w and the total value v of this selection in the node, along with some upper bound ub on the value of any subset that can be obtained by adding zero or more items to this selection.

Item	Weight	value	value / weight	capacity
1	4	$40	10	
2	7	$42	6	$W = 10$
3	5	$25	5	
4	3	$1 2	4	
	w=19	v=119	$v_{i+1}/w_{i+1}=25$	

Asimple way to compute the upper bound ub is to add to v, the total value of the items already selected, the product of the remaining capacity of the knapsack $W - w$ and the best per unit payoff among the remaining items, which is v_{i+1}/w_{i+1}:

$$ub = v + (W - w)(v_{i+1}/w_{i+1}).$$
$$= 0+(10\text{-}0)\,(10)$$
$$= 100$$
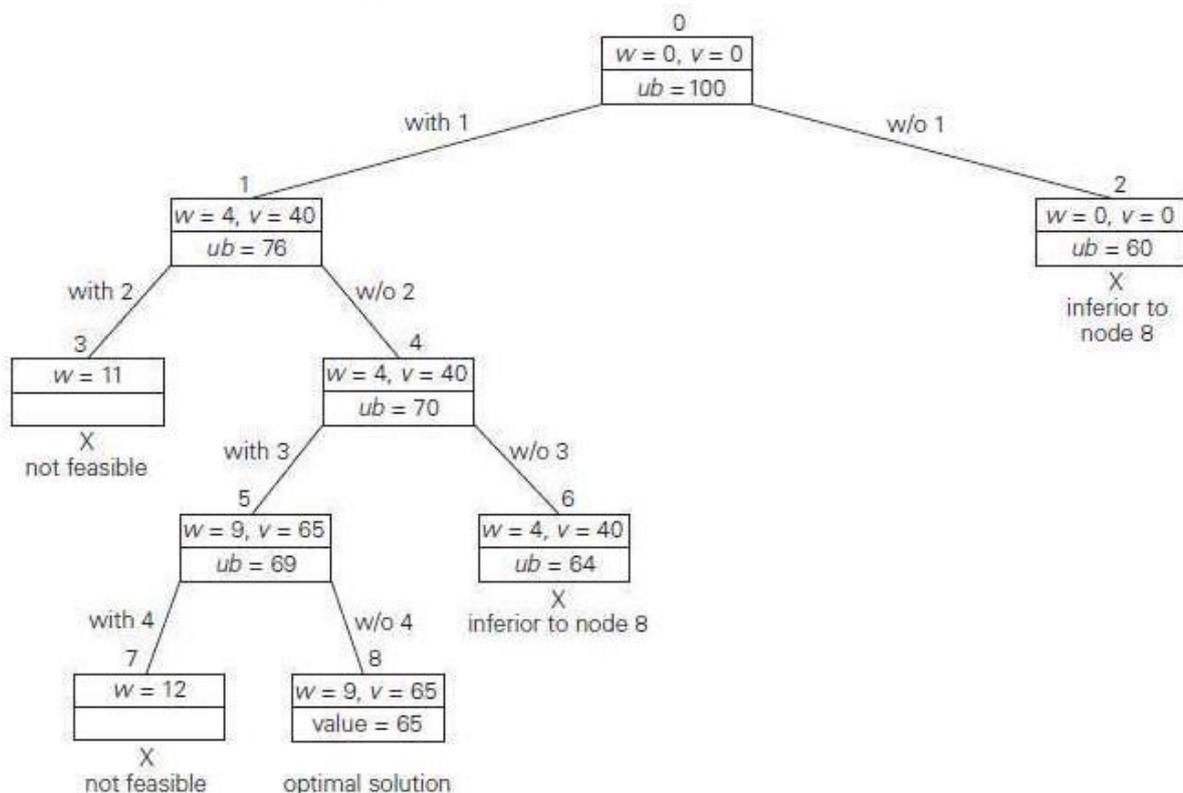
FIGURE 5.18 State-space tree of the best-first branch-and-bound algorithm for the instance of the knapsack problem.

At the root of the state-space tree (see Figure 5.18), no items have been selected as yet. Hence, both the total weight of the items already selected w and their total value v are equal to 0. The value of the upper bound computed by formula (12.1) is \$100. Node 1, the left child of the root, represents the subsets that include item 1. The total weight and value of the items already included are 4 and \$40, respectively; the value of the upper bound is $40 + (10 - 4) * 6 = \$76$. Node 2 represents the subsets that do not include item 1. Accordingly, $w = 0$, $v = \$0$, and $ub = 0 + (10 - 0) * 6 = \$60$. Since node 1 has a larger upper bound than the upper bound of node 2, it is more promising for this maximization problem, and we branch from node 1 first. Its children—nodes 3 and 4—represent subsets with item 1 and with and without item 2, respectively.

Since the total weight w of every subset represented by node 3 exceeds the knapsack's capacity, node 3 can be terminated immediately. Node 4 has the same values of w and v as its parent; the upper bound ub is equal to $40 + (10 - 4) * 5 = \$70$. Selecting node 4 over node 2 for the next branching (why?), we get nodes 5 and 6 by respectively including and excluding item 3. The total weights and values as well as the upper bounds for these nodes are computed in the same way as for the preceding nodes. Branching from node 5 yields node 7, which represents no feasible solutions, and node 8, which represents just a single subset {1, 3} of value \$65. The remaining live nodes 2 and 6 have smaller upper-bound values than the value of the solution represented by node 8. Hence, both can be terminated making the subset {1, 3} of node 8 the optimal solution to the problem.

Solving the knapsack problem by a branch-and-bound algorithm has a rather unusual characteristic. Typically, internal nodes of a state-space tree do not define a point of the problem's search space, because some of the solution's components remain undefined. If we had done this for the instance investigated above, we could have terminated nodes 2 and 6 before node 8 was generated because they both are inferior to the subset of value \$65 of node 5.

5.13 TRAVELING SALESMAN PROBLEM

We will be able to apply the branch-and-bound technique to instances of the travelling salesman problem if we come up with a reasonable lower bound on tour lengths. One very simple lower bound can be obtained by finding the smallest element in the intercity distance matrix D and multiplying it by the number of cities n. But there is a less obvious and more informative lower bound for instances with symmetric matrix D, which does not require a lot of work to compute. It is not difficult to show (Problem 8 in this section's exercises) that we can compute a lower bound on the length l of any tour as follows. For each city i, $1 \le i \le n$, find the sum s_i of the distances from city i to the two nearest cities; compute the sum s of these n numbers, divide the result by 2, and, if all the distances are integers, round up the result to the nearest integer:

$$lb = \lceil s/2 \rceil$$
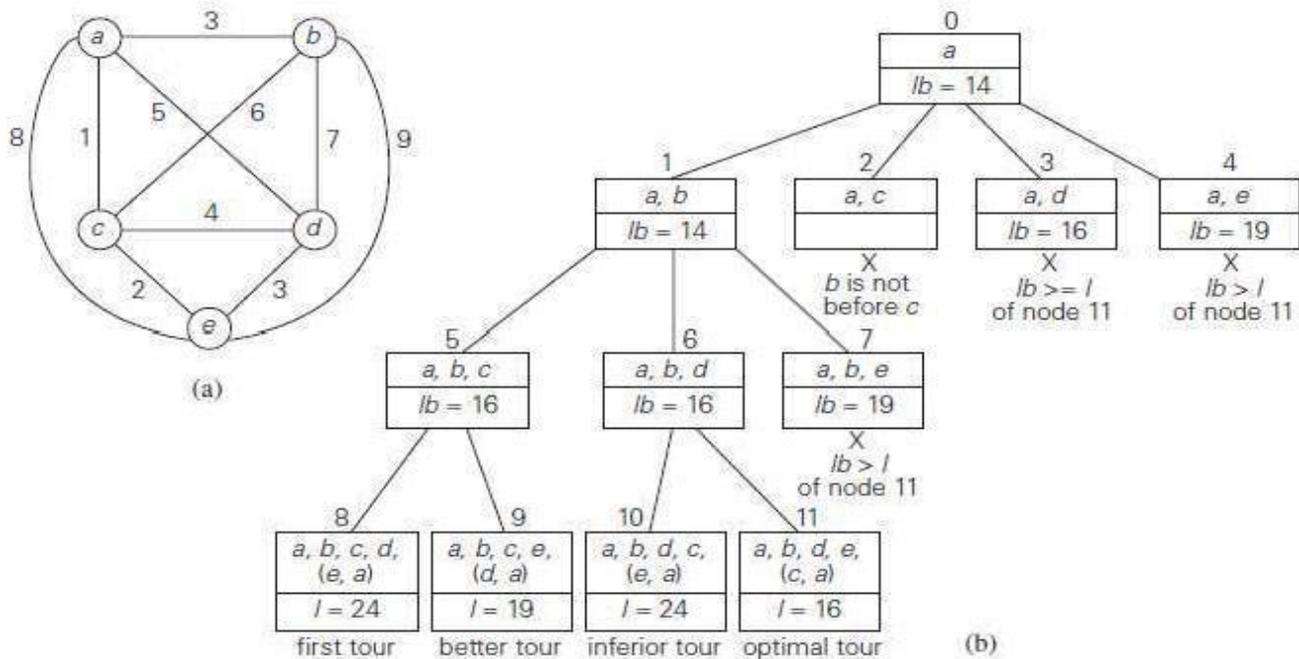
FIGURE 5.19 (a)Weighted graph. (b) State-space tree of the branch-and-bound algorithm to find a shortest Hamiltonian circuit in this graph. The list of vertices in a node specifies a beginning part of the Hamiltonian circuits represented by the node.

For example, for the instance in Figure and above formula yields
$lb = [[(1 + 3) + (3 + 6) + (1 + 2) + (3 + 4) + (2 + 3)]/2] = 14$.

Moreover, for any subset of tours that must include particular edges of a given graph, we can modify lower bound accordingly. For example, for all the Hamiltonian circuits of the graph in Figure that must include edge (a, d), we get the following lower bound by summing up the lengths of the two shortest edges incident with each of the vertices, with the required inclusion of edges (a, d) and (d, a):

$[[(1 + 5) + (3 + 6) + (1 + 2) + (3 + 5) + (2 + 3)]/2] = 16$.

We now apply the branch-and-bound algorithm, with the bounding function given by formula, to find the shortest Hamiltonian circuit for the graph in Figure 5.19a. To reduce the amount of potential work. First, without loss of generality, we can consider only tours that start at a. Second, because our graph is undirected, we can generate only tours in which b is visited before c. In addition, after visiting $n - 1 = 4$ cities, a tour has no choice but to visit the remaining unvisited city and return to the starting one. The state-space tree tracing the algorithm's application is given in Figure 5.19b.

Branch and Bound – 15 – Puzzle Problem

The 15 puzzle problem is a well-known sliding puzzle game where a player has to move tiles numbered from 1 to 15 on a 4x4 grid to reach a target configuration. The goal is to reach the target configuration in the minimum number of moves possible.

Branch and Bound is a search algorithm that solves combinatorial optimization problems by systematically exploring the space of possible solutions. It is often used to solve the 15 puzzle problem and other similar problems.

The Branch and Bound algorithm works by exploring the search space in a systematic way, keeping track of the best solution found so far, and pruning branches of the search tree that are guaranteed to lead to suboptimal solutions.

To apply the Branch and Bound algorithm to the 15 puzzle problem, we start with the initial configuration of the puzzle and generate all possible moves from that configuration. Each move generates a new configuration, and we add these new configurations to a priority queue ordered by the estimated distance from the target configuration.

We then pick the configuration with the lowest estimated distance from the target configuration and explore its children, repeating this process until we reach the target configuration.

As we explore the search space, we keep track of the best solution found so far and prune branches that are guaranteed to lead to worse solutions than the best solution found so far. This is done by maintaining an upper bound on the cost of the best solution that can be found in each branch of the search tree.

By combining the Branch and Bound algorithm with efficient heuristics for estimating the distance to the target configuration, it is possible to find optimal solutions to the 15 puzzle problem and other similar problems in reasonable time.

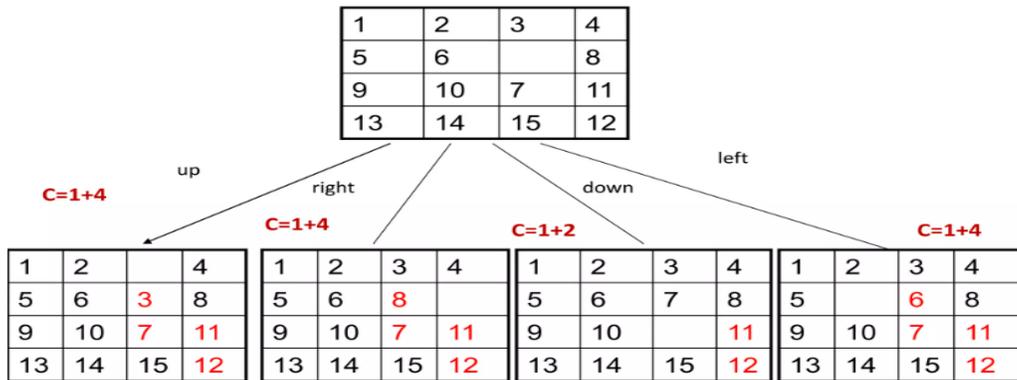| 1	2	3	4
5	6		8
9	10	7	11
13	14	15	12

Initial arrangement

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Goal arrangement

Stage 1:

1	2	3	4
5	6		8
9	10	7	11
13	14	15	12

up

right

down

left

C=1+4

C=1+4

C=1+2

C=1+4

1	2		4
5	6	3	8
9	10	7	11
13	14	15	12

1	2	3	4
5	6	8	
9	10	7	11
13	14	15	12

1	2	3	4
5	6	7	8
9	10		11
13	14	15	12

1	2	3	4
5		6	8
9	10	7	11
13	14	15	12

Stage 2:

1	2		4
5	6	3	8
9	10	7	11
13	14	15	12

1	2	3	4
5	6	8	
9	10	7	11
13	14	15	12

1	2	3	4
5	6	7	8
9	10		11
13	14	15	12

1	2	3	4
5		6	8
9	10	7	11
13	14	15	12

C=2+1

C=2+3

C=2+3

1	2	3	4
5	6	7	8
9	10	11	
13	14	15	12

1	2	3	4
5	6	7	8
9	10	15	11
13	14		12

1	2	3	4
5	6	7	8
9		10	11
13	14	15	12

C=3+2

C=3+0

1	2	3	4
5	6	7	8
9	10		11
13	14	15	12

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

UNIT V

NP-COMPLETE AND APPROXIMATION ALGORITHM 9

Tractable and intractable problems: Polynomial time algorithms – Venn diagram representation – NPalgorithms – NP-hardness and NP-completeness – Bin Packing problem – Problem reduction: TSP –3CNF problem. Approximation Algorithms: TSP – Randomized Algorithms: concept and application- Primality testing – randomized quick sort – Finding kth smallest number

Tractable and Intractable Problems

In computer science, the classification of problems into tractable and intractable is a crucial aspect ofalgorithm design and analysis. A tractable problem is one that can be solved efficiently, while an intractable problem is one that cannot be solved efficiently, at least not with our current understandingof computer algorithms.

For instance, the problem of finding the greatest common divisor of two integers is a tractable problem, since there is a simple and efficient algorithm to solve it, namely the Euclidean algorithm. On the other hand, the problem of factoring a large composite number into its prime factors is currently believed to be an intractable problem, meaning that there is no known algorithm that can solve it efficiently for largenumbers.

The distinction between tractable and intractable problems is important because it allows us to focusour algorithmic efforts on the problems that can be solved efficiently and to avoid wasting time on problems that are likely to be intractable.

Venn Diagrams Representation

Venn diagrams are a type of diagram used to represent sets and their relationships. They consist of a rectangle representing the universe of discourse, and circles or other shapes representing sets withinthat universe. The diagram can be used to show the relationships between sets, such as their
intersections or unions.

For example, consider the sets A and B where A={1,2,3}, B={2,3,5,6}. A Venn diagram for these sets would consist of two circles, one for each set, with the regions of intersection showing the elements thatare in more than one set. The diagram would look like a collection of overlapping circles, with the regionof overlap showing the elements that are In all three sets.
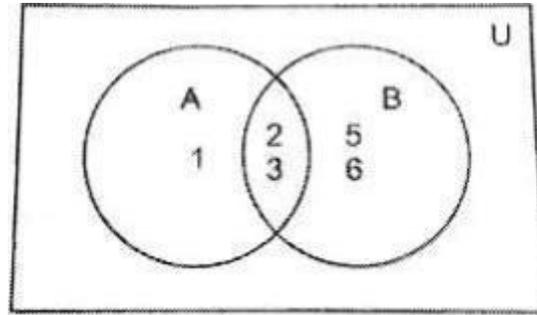
Venn diagrams are useful for visualizing set relationships and can be used in a variety of contexts, frombasic mathematics to data analysis and statistics.

Polynomial Time Algorithms

A polynomial time algorithm is one that can solve a problem in time that is proportional to some polynomial function of the size of the input. For example, an algorithm that takes time $O(n^2)$ to sort anarray of n numbers is a polynomial time algorithm.

Polynomial time algorithms are important because they are typically considered to be efficient. While the running time of a polynomial time algorithm may still grow quite large for very large input sizes, thegrowth rate is generally manageable, and the algorithm can be used to solve problems that are too largefor brute force methods but too small to require more specialized algorithms.

NP Algorithms
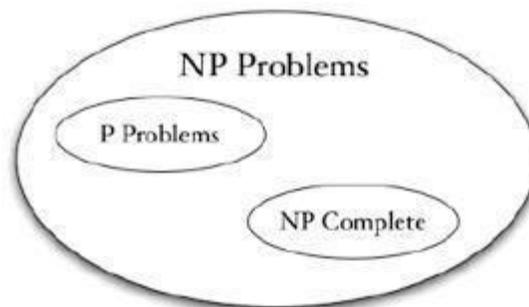
NP (nondeterministic polynomial time) algorithms are a class of algorithms that can solve problems in polynomial time on a theoretical computer that can perform a certain type of nondeterministic computation. These algorithms are often used to solve problems that are difficult or impossible to solveefficiently with deterministic algorithms.

For example, the problem of finding a Hamiltonian cycle in a graph is an NP problem, meaning that no known algorithm can solve it efficiently for all instances of the problem. However, there is an NP algorithm called the traveling salesman algorithm that can solve the problem efficiently for many instances.

NP algorithms are important because they allow us to solve many important problems that would otherwise be intractable. However, they are generally not considered to be efficient in practice, since they require significant computational resources and are often difficult to implement.

NP-Hard and NP-Completeness
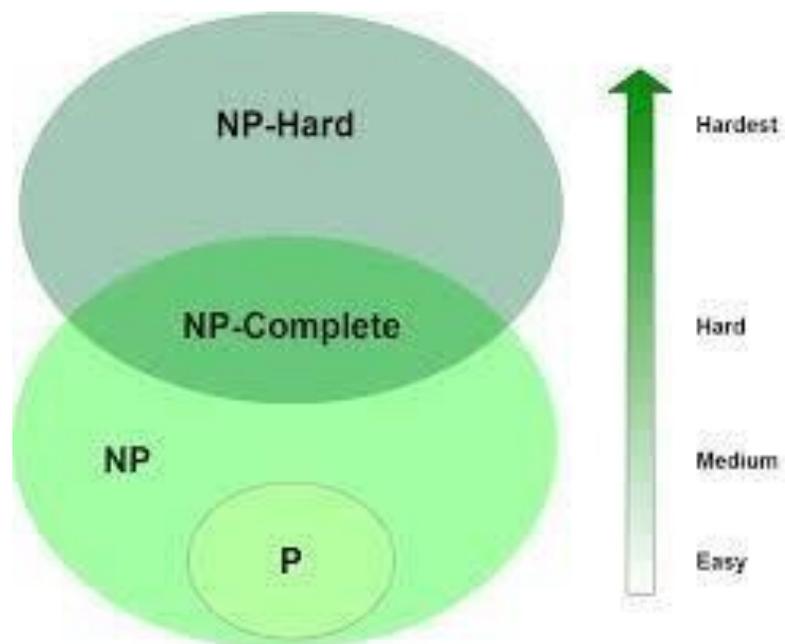
NP-hard and NP-complete problems are a class of problems that are at least as difficult as NP problems, meaning that they are likely to be intractable

NP-hard problems are those that can be reduced to any other NP problem in polynomial time. In other words, if we can solve

Reduction of np problem to NP-Hard and np-complete

Reduction is a technique used in computational complexity theory to show that one problem is no harder than another. Specifically, it involves demonstrating that a problem A can be transformed into

problem B in such a way that any solution to problem B can be used to solve problem A. In this answer, we will focus on the reduction of NP problems to NP-hard and NP-complete problems, which are two
important classes of problems in computational complexity theory

The steps involved in reducing an NP problem to an NP-hard problem:

Step 1: Choose an NP-hard problem, such as the Knapsack Problem, that you want to reduce your NP problem to.

Step 2: Construct a reduction algorithm that takes an instance of your NP problem and converts it into an instance of the NP-hard problem. This algorithm should run in polynomial time.

Step 3: Prove that the reduction algorithm is correct by showing that any solution to the NP-hard problem can be used to solve the original NP problem.

Step 4: Prove that the reduction algorithm runs in polynomial time by showing that it takes a polynomial amount of time to transform an instance of the original problem into an instance of the NP-hard problem

Bin Packing Problem

The bin packing problem is a classic optimization problem in which items of varying sizes must be packed into a set of containers, or "bins", in a way that minimizes the number of bins required. The problem can be stated as follows:

Given a set of items with sizes s1, s2, …, sn, and a set of bins with capacity C, find the minimum number of bins required to pack all the items, subject to the constraint that the sum of the sizes of the items in each bin must not exceed the bin capacity.

The bin packing problem Is known to be NP-hard, which means that there is no known algorithm that can solve it efficiently for all cases. However, there are a number of approximation algorithms and heuristics that can be used to find near-optimal solutions in a reasonable amount of time. These algorithms typically involve sorting the items by size and then iteratively adding them to bins in a way that minimizes the number of bins required.

There are several types of bin packing algorithms, including:

- First Fit (FF) algorithm: This algorithm places items into the first available bin that can accommodate them, starting from the first bin.

- Next Fit (NF) algorithm: This algorithm is similar to the First Fit algorithm, but instead of starting from the first bin every time, it starts from the last bin used in the previous iteration.

- Best Fit (BF) algorithm: This algorithm places items into the bin that has the least amount of extra space left after accommodating the item.

- Worst Fit (WF) algorithm: This algorithm places items into the bin that has the most amount of extra space left after accommodating the item.

OFFLINE ALGORITHMS

- First Fit Decreasing (FFD) algorithm: This algorithm first sorts the items in decreasing order of size, and then applies the First Fit algorithm.

- Best Fit Decreasing (BFD) algorithm: This algorithm first sorts the items in decreasing order of size, and then applies the Best Fit algorithm.

- Largest Area Fit (LAF) algorithm: This algorithm considers the area of the remaining space in the bin and places the item in the bin that has the largest remaining area.

Each algorithm has its own advantages and disadvantages, and the best algorithm to use depends on the specific requirements and constraints of the problem at hand.

Applications: The bin packing problem has a wide range of applications in logistics, transportation, and manufacturing, where it is often necessary to pack items efficiently to minimize shipping costs or maximize warehouse space utilization.

TSP and 3CNF Reduction

The Traveling Salesman Problem (TSP) and the 3-Satisfiability Problem (3CNF) are two well-known problems in computer science. The TSP is an optimization problem that seeks to find the shortest possible route that visits a set of cities and returns to the starting city. The 3CNF problem is a decision problem that asks whether there is a satisfying assignment of variables that satisfies a set of clauses in conjunctive normal form (CNF) where each clause contains exactly three literals.

To reduce TSP to 3CNF, we can start by assuming that we have a TSP instance with a set of cities and distances between each pair of cities. We can then construct a set of boolean variables that correspond to each possible route between the cities. For each variable, we can assign the value "true" if the
corresponding route is included in the optimal TSP tour and "false" otherwise.

Next, we need to construct a set of clauses that encode the constraints of the TSP problem. Specifically, we need to ensure that each city is visited exactly once, that the tour is closed (i.e., it returns to the starting city), and that the length of the tour is minimized. To encode these constraints, we can construct clauses that ensure that each city is visited exactly once, that each city is visited by exactly one incoming and one outgoing edge, and that the sum of the lengths of the edges in the tour is minimized.

To reduce 3CNF to TSP, we can start by assuming that we have a 3CNF instance with a set of boolean variables and a set of clauses. We can then construct a set of cities that correspond to each possible assignment of the boolean variables. For each assignment, we can assign a distance of 1 if the
assignment satisfies the clause and a distance of 2 otherwise.

Next, we need to construct a distance matrix that encodes the distances between each pair of cities. Specifically, we can assign a distance of 0 between two cities if they correspond to the same assignment of variables, a distance of 1 if they differ by one variable assignment, and a distance of 2 if they differ by two or more variable assignments.

Finally, we can solve the TSP problem on this distance matrix to find the shortest possible tour that satisfies all of the clauses in the 3CNF instance. If the length of the tour is equal to the number of assignments that satisfy all of the clauses, then the 3CNF instance is satisfiable. Otherwise, it is unsatisfiable.

Randomized algorithms

Randomized algorithms are algorithms that use randomness, typically through the generation of random numbers, to solve computational problems. They are particularly useful in situations where a
deterministic algorithm may take too much time or memory to solve a problem, or where there is no known deterministic algorithm that can solve the problem efficiently.

There are two main types of randomized algorithms:

- Las Vegas algorithms

- Monte Carlo algorithms

Las Vegas algorithms

Las Vegas algorithms are randomized algorithms that always return the correct answer, but the running time can vary depending on the input. An example of a Las Vegas algorithm is quicksort, where the pivot element is chosen randomly and the expected running time is O(nlogn).

Monte Carlo algorithms

Monte Carlo algorithms, on the other hand, may return an incorrect answer with a certain probability, but the running time is typically fixed. They are often used in situations where the cost of obtaining a correct answer is very high, or where a probabilistic answer is sufficient. An example of a Monte Carlo algorithm is the Miller-Rabin primality test, which is used to determine whether a number is prime or composite with a certain probability.

Applications: Randomized algorithms are used in a wide range of applications, including cryptography, machine learning, computational biology, and optimization. They have been shown to be very effective in practice, and have led to many breakthroughs in computer science and related fields.

Primality testing

Primality testing is the process of determining whether a given number is prime or composite. A prime number is a positive integer greater than 1 that has no positive integer divisors other than 1 and itself.

There are several types of primality testing algorithms, each with different time and space complexity,and different tradeoffs between speed and accuracy.

- •Trial division: This is the simplest and most straightforward primality testing algorithm. It involves checking whether the number is divisible by any integer between 2 and the square root of the number. Ifno divisor is found, the number is prime. However, this algorithm is slow for large numbers.

For example, let's say we want to test whether the number 37 is a prime number using trial division. Thefirst prime number is 2, and the square root of 37 is approximately 6.08. Therefore, we only need to test divisibility by 2, 3, and 5.

- •37 is not divisible by 2, since it is an odd number.

- •37 is not divisible by 3, since the sum of its digits (3 + 7) is not divisible by 3.

- •37 is not divisible by 5, since it does not end in 0 or 5.

Since we have tested all the primes less than or equal to the square root of 37, and none of them divide37 evenly, we can conclude that 37 is a prime number.

- •Fermat's little theorem: This theorem states that if p is a prime number and a is any positive integer not divisible by p, then a raised to the power of p-1 is congruent to 1 modulo p. This means that if we pick a random value of a and check whether a^(p-1) is congruent to 1 modulo p, we can determine with high probability whether p is prime or not. However, this algorithm can give false positives for certain composite numbers that satisfy the theorem.

Here's an example: let's say we want to test if n = 17 is prime. We can choose a few random integers, saya = 2, 3, and 5, and check if:

2^(16) mod 17 = 1

3^(16) mod 17 = 1

5^(16) mod 17 = 1

In this case, all three values are equal to 1, so we can conclude that 17 is likely to be prime. However, it'simportant to note that this is only a preliminary test and does not provide absolute certainty that n is prime. More rigorous methods, such as the Miller-Rabin test or AKS algorithm, may be needed for largernumbers.

- •Miller-Rabin: This algorithm is a probabilistic primality testing algorithm that is based on the same idea as Fermat's little theorem. It involves picking a random value of a and checking whether a^(n-1) is congruent to 1 modulo n, where n is the number we want to test for primality. If not, we keep raising a to successive powers until we find a value of k such that a^(2^k * (n-1)) is congruent to 1 modulo n. If k is0 or a power of 2, then n is probably prime. Otherwise, n is composite. This algorithm has a very high probability of correctly identifying primes, and can be made arbitrarily accurate by running multiple tests with different random values of a.

Example:Suppose we want to test if the number 27 is prime or composite using the Miller-Rabin algorithm.

Step 1: Write 27-1 as 26 = 2^1 * 13.

Step 2: Choose a random base, say a=2.

Step 3: Compute a^d mod n for d = 13, 6, 3, and 2.

A^d mod n = 2^13 mod 27 =

2 A^(d/2) mod n = 2^6 mod

27 = 1A^(d/4) mod n = 2^3

mod 27 = 8A^(d/8) mod n =

2^2 mod 27 = 4

Step 4: Check if any of the computed values are equal to n-1 = 26.

In this case, none of the computed values are equal to 26, so we can conclude that 27 is composite.

Note: The Miller-Rabin algorithm is a probabilistic algorithm, which means that there is a small chance that it might give a false positive result (i.e., declare a composite number as prime). The probability of a false positive can be made arbitrarily small by repeating the test with different random bases.

- AKS: This algorithm is a deterministic primality testing algorithm that was discovered in 2002. It is based on a polynomial-time algorithm for testing whether two polynomials are identical. By using this algorithm to test whether certain polynomials are identical modulo n, we can determine with certainty whether n is prime or composite. However, this algorithm is relatively slow compared to the probabilistic algorithms, and is only practical for relatively small numbers.

Example:

$$n \text{ prime} \Rightarrow a^n \equiv a \pmod{n}$$

- Let n = 91
    - Composite: 91 = 7 * 13

- $3^{91} \equiv 3 \pmod{91}$
    - 91 is a Fermat pseudoprime base 3

- $2^{91} \neq 2 \pmod{91}$
    - 91 is not a Fermat pseudoprime base 2 (91 is composite)

Randomized Quick Sort

Randomized Quick Sort is a popular sorting algorithm that uses the divide and conquer approach. It is an efficient algorithm that sorts an array in ascending or descending order by selecting a pivot element and partitioning the array into two parts: the left part containing elements less than the pivot, and the right

part containing elements greater than or equal to the pivot. The algorithm then recursively sorts the leftand right parts until the entire array is sorted.

The algorithm works as follows:

1. Choose a random pivot element from the array.

2. Partition the array into two parts: the left part containing elements less than the pivot, and the right part containing elements greater than or equal to the pivot.

3. Recursively sort the left and right parts using the same algorithm until the entire array is sorted.

Example: Consider the following array of integers: [5, 3, 7, 2, 8, 4, 1, 6]

Step 1: Choose a random pivot element from the array. Let's say we choose 4.

Step 2: Partition the array into two parts: the left part containing elements less than the pivot (3, 2, 1),and the right part containing elements greater than or equal to the pivot (5, 7, 8, 6).

Step 3: Recursively sort the left and right parts using the same algorithm.

★Sorting the left part [3, 2, 1]:

• Choose a random pivot element from the array. Let's say we choose 2.

• Partition the array into two parts: the left part containing elements less than the pivot (1), and the right part containing elements greater than or equal to the pivot (3, 2).

• Recursively sort the left and right parts using the same algorithm.

★Sorting the left part [1]:

The array is already

sorted.Sorting the right

part [3, 2]:

• Choose a random pivot element from the array. Let's say we choose 3.

• Partition the array into two parts: the left part containing elements less than the pivot (2), and the right part containing elements greater than or equal to the pivot (3).

• Recursively sort the left and right parts using the same algorithm.

★Sorting the left part [2]:

The array is already sorted.

★Sorting the right part [3]:

The array is already sorted.

Therefore, the left part [3, 2, 1] is sorted to [1, 2, 3].

★Sorting the right part [5, 7, 8, 6]:

- Choose a random pivot element from the array. Let's say we choose 7.

- Partition the array into two parts: the left part containing elements less than the pivot (5, 6), and the right part containing elements greater than or equal to the pivot (7, 8).

- Recursively sort the left and right parts using the same algorithm.

★Sorting the left part [5, 6]:

The array is already sorted.

★Sorting the right part [7, 8]:

The array is already sorted.

Therefore, the right part [5, 7, 8, 6] is sorted to [5, 6, 7, 8].

Finally, we concatenate the sorted left and right parts to obtain the sorted array: [1, 2, 3, 4, 5, 6, 7, 8]

Finding kth smallest number

The problem of finding the kth smallest element in an unsorted array or list is a common problem in computer science and is often used in design analysis and algorithms. The goal is to find the kth smallestelement, where k is a positive integer and is less than or equal to the length of the array or list.

One common solution to this problem is to use a sorting algorithm to sort the array or list in ascending order, and then return the element at the kth position. This solution has a time complexity of O(nlogn) inthe worst case, where n is the size of the array or llist

Another solution to this problem is to use a selection algorithm. A selection algorithm is an algorithm that finds the kth smallest element in an unsorted array or list without sorting the entire array or list. One example of a selection algorithm is the Quickselect algorithm, which is an efficient and commonly used algorithm for this problem. The Quickselect algorithm has a time complexity of O(n) in the averagecase, where n is the size of the array or list.

The Quickselect algorithm works by selecting a pivot element from the array or list, partitioning the arrayor list into two subarrays, and recursively selecting a new pivot element from the subarray that containsthe kth smallest element. The algorithm repeats this process until the kth smallest element is found.

Here are the steps for the Quickselect algorithm:

1. Select a pivot element from the array or list.

2. Partition the array or list into two subarrays: one subarray contains all elements less than the pivot, and the other subarray contains all elements greater than or equal to the pivot.

3. If the kth smallest element is in the subarray that contains elements less than the pivot, recursively apply the algorithm to that subarray.

4. If the kth smallest element is in the subarray that contains elements greater than or equal to the pivot, recursively apply the algorithm to that subarray.

5. If the pivot is the kth smallest element, return the pivot.

6. If the kth smallest element is not in either subarray, return an error.

The Quickselect algorithm has a worst-case time complexity of O(n^2) in the worst case, but this can be avoided by selecting a good pivot element. In practice, the Quickselect algorithm has been found to be very efficient and is often used in applications where finding the kth smallest element is a common operation.

Example:

- Input: [7, 4, 6, 3, 9, 1] k = 2.
- Output: k'th smallest aííay element is 3.
- Input: [7, 4, 6, 3, 9, 1] k = 1.
- Output: k'th smallest aííay element is 1.

C

CS 3401- ALGORITHMS

UNIT I
INTRODUCTION

Algorithm analysis: Time and space complexity - Asymptotic Notations and its properties Best case, Worst case and average case analysis – Recurrence relation: substitution method - Lower bounds – **searching:** linear search, binary search and Interpolation Search, **Pattern search:** The naïve string- matching algorithm - Rabin-Karp algorithm - Knuth-Morris-Pratt algorithm. **Sorting:** Insertion sort – heap sort

PART - A

1. **What do you mean by algorithm? (Nov/Dec 2008) (May/June 2013) (Apr/May 17) (U)**
   An algorithm is a sequence of unambiguous for solving a problem i.e., for obtaining a required output for any legitimate input in a finite amount of time. In addition, all algorithms must satisfy the following criteria:
   1) Input
   2) Output
   3) Definiteness
   4) Finiteness
   5) Effectiveness.

2. **What is performance measurement? (R)**
   Performance measurement is concerned with obtaining the space and the time requirements of a particular algorithm.

3. **Give the diagram representation of Notion of algorithm. (C)**

PROBLEM

↓

ALGORITHM

INPUT ⟶ COMPUTER ⟶ OUTPUT
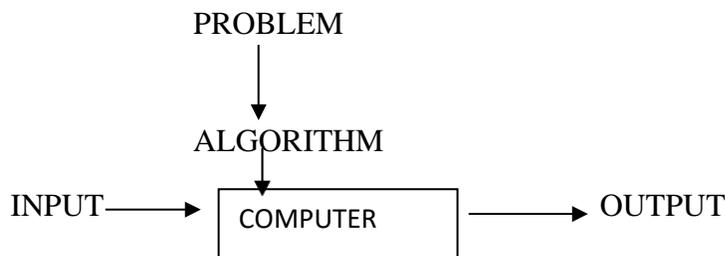
4. **What are the types of algorithm efficiencies? (R)**
   The two types of algorithm efficiencies are
   Time efficiency: indicates how fast the algorithm runs.
   Space efficiency: indicates how much extra memory the algorithm needs.

5. **What is space complexity? (Nov/Dec 2012) (R)**
   Space Complexity indicates how much extra memory the algorithm needs. Basicallyit has three components. They are instruction space, data space and environment space.

6. **What is time complexity? (Nov/Dec 2012) (R)**
   Time Complexity indicates how fast an algorithm runs. T (P) =Compile Time + Run Time.(Tp) ,Where Tp is no of add, sub, mul...

7. **What is an algorithm design technique? (R)**
   An algorithm design technique is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing.

8. **What is pseudo code? (R)**
   A pseudo code is a mixture of a natural language and programming language constructs to specify an algorithm. A pseudo code is more precise than a natural language and its usage often yields more concise algorithm descriptions.

9. **What are the types of algorithm efficiencies? (R)**
   The two types of algorithm efficiencies are
   Time efficiency: indicates how fast the algorithm runs
   Space efficiency: indicates how much extra memory the algorithm needs.

10. **What do you mean by "worst-case efficiency" of and algorithm? (R) (Nov 17)**
    The worst case efficiency of an algorithm, its efficiency for the worst-case input of size n, which is an input or inputs of size n for which the algorithm runs the longest among all possible inputs of that size.

11. **What is best-case efficiency? (R)**
    The best-case efficiency of an algorithm is its efficiency for the best-case input of size n, which is an input or inputs for which the algorithm runs the fastest among all possible inputs of that size.

12. **What is average case efficiency? (May 2008) (R)**
    The average case efficiency of an algorithm is its efficiency for an averagecase input of size n. It provides information about an algorithm behavior on a "typical" or "random" input.

13. **Define asymptotic notations. (R)**
    To choose best algorithm, we need to check efficiency of each algorithm the efficiency can be measured by computing time complexity of each algorithm. Asymptotic notation is shorthand way to represent the time complexity
    The various notations are Big "Oh", Big Omega, and Theta.

14. **Define the asymptotic notation "Big oh" (0)(May 2008)(April/May 2012&2013) (R)**
    Let, f(n) and g(n) be two non-negative functions. Let, n0 and constant c are two integers such that no denotes some value of input similarly c is constant such that c > 0. We can write
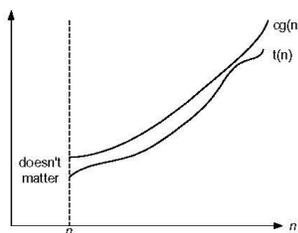
F(n) <= c*g(n) For all n ≥ n0
A function t(n) is said to be in O(g(n)) denoted t(n) ε O (g(n)), if t(n) is bounded above by some constant multiple of g(n) for all large n, i.e., if there exist some positive constant c and some non-negative integer n0 such that
T (n) < c g (n) for n > n0

15. Define the asymptotic notation "Omega" (Ω). (R)

A function f(n) is said to be in Ω (g(n)) if f(n) is bounded below by some positive constant multiple of g(n) such that
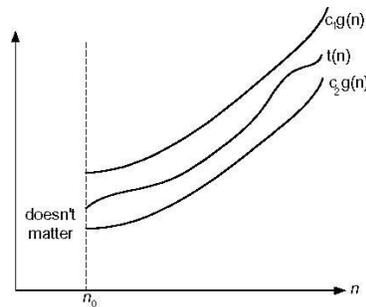$$f(n) \geq c * g(n) \text{ For all } n \geq n0$$
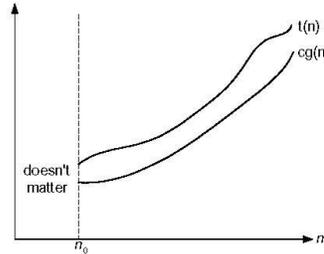
16. Define the asymptotic notation "theta" (θ) (R)

Let, f(n) and g(n) be two non negative functions. There are two positive constants namely c1 and c2 such that c1 ≤ g(n) ≤ c2 g(n)
Then we can say that, f(n) ε Θ (g(n))

17. What is recursive algorithm? (R)

An algorithm is said to be recursive if the same algorithm is invoked in the body. An algorithm that calls itself is direct recursive.

18. Define recurrence. (May2010) (R)

A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs. The complexity of many interesting algorithms is easily expressed as a recurrence, especially divide and conquer algorithms. The complexity of recursive algorithms is readily expressed as a recurrence.
    Example :(i)Linear search of a list

$$T(n) = \begin{cases} 1 & \text{for } n \leq 1 \\ T(n-1) + 1 & \text{otherwise} \end{cases}$$
        (ii) Binary search

$$T(n) = \begin{cases} 1 & \text{for } n \le 1 \\ T(n/2) + 1 & \text{otherwise} \end{cases}$$

19. Define Linear Search. (Nov/Dec 2011) (April/May 2010&2012) (R)

In computer science, linear search or sequential search is a method for finding aparticular value in a list, which consists in checking every one of its elements, one at atime and in sequence, until the desired one is found.

20. What are the Best, Worst and Average Case complexity of Linear Search? (R)
- ✓ Best Case – O(1)
- ✓ Average Case – O(N)
- ✓ Worst Case – O(N)

21. Difference between Best Case and Worst Case Complexities.(AN)

The best case complexity of an algorithm is its efficiency for the best case input ofsize N, which is an input of size N for which the algorithm runs fastest among all possible inputs of that size.

The worst case complexity of an algorithm is its efficiency for the worst case inputof size N, which is an input of size N for which the algorithm runs longest among all possible inputs of that size.

22. What is binary search? (R)

Binary search is a remarkably efficient algorithm for searching in a sorted array. It works by comparing a search key K with the arrays middle element A[m]. if they match the algorithm stops; otherwise the same operation is repeated recursively for the first half of the array if K < A[m] and the second half if K > A[m].

    a. A[0]………A[m-1] A[m] A[m+1]………A[n-1]

23. Give computing time for Binary search?(APRIL/MAY 2012) (E)

In conclusion we are now able completely describe the computing time of binary search by giving formulas that describe the best, average and worst cases.

Successful searches

Best-(1) average-(logn) worst -(Logn)

unsuccessful searches best, average, worst- (logn)

24. Write the algorithm for Iterative binary search? (A)

```
Algorithm BinSearch(a,n,x)
//Given an array a[1:n] of elements in nondecreasing
// order, n>0, determine whether x is present
{
low : = 1;
high : = n;
while (low < high) do
{
mid : = [(low+high)/2];
if(x □□a[mid]) then high:= mid-1;
else if (x □a[mid]) then low:=mid + 1;
```

```
 else return mid;
 }
 return 0;
 }
```

25. Give the general plan for analyzing non recursive algorithm. (R)
- Decide a parameter indicating an Input's Size.
- Identify the algorithms Basic Operation
- Check whether the number of times the basic operation is executed only on thesize of an input. If it also depends on some additional property, the worst case, average case, and if necessary, best case efficiencies have to be investigated separately.
- Using standard formulas and rules of sum manipulation either find a closedformula for the count or, at the very least, establish its order of growth.

26. What is validation of algorithm? (Nov/Dec 2012) (R)
The process of measuring the effectiveness of an algorithm before it is coded to know the algorithm is correct for every possible input. This process is called validation.

27. What are all the methods available for solving recurrence relations? (R)
- ✓ Forward Substitution
- ✓ Backward Substitution
- ✓ Smoothness Rule
- ✓ Master Theorem

28. Write down the problem types.(April/May 2008) (R)
- ✓ Sorting
- ✓ Searching
- ✓ String Processing
- ✓ Graph Problem
- ✓ Combinational Problem
- ✓ Geometric Problem
- ✓ Numerical Problem

29. What are the types of recurrence relations? (R)
- ✓ Homogeneous recurrence relation.
- ✓ Non homogeneous recurrence relation.

30. Define Substitution Method. (April /May 2010) (R)
Substitution method is a method of solving a system of equations wherein one of theequations is solved for one variable in terms of the other variables.

31. Give the general plan for analyzing recursive algorithm. (R)
- ✓ Decide a parameter indicating an Input's Size.
- ✓ Identify the algorithms Basic Operation

✓ Check whether the number of times the basic operation is executed only on thesize of an input. If it also depends on some additional property, the worst case,average case, and if necessary, best case efficiencies have to be investigatedseparately.

✓ Set up a recurrence relation, with an appropriate initial condition, for thenumber of times basic operation is executed.

32. Define Recurrence Relation. (APRIL/MAY 2010) (Nov/Dec 2016) (R)
The equation defines M (N) not explicitly, ie.., is a function of n, but implicitly as afunction of its value at another point, namely N-1. Such equations are calledrecurrence relation.

33. Give the General Plan for the Empirical Analysis of Algorithm Time Efficiency? (C)
1. Understand the experiment's purpose.
2. Decide on the efficiency metric M to be measured and the measurement unit (an operation count vs. a time unit).
3. Decide on characteristics of the input sample (its range, size, and so on).
4. Prepare a program implementing the algorithm (or algorithms) for the experimentation.
5. Generate a sample of inputs.
6. Run the algorithm (or algorithms) on the sample's inputs and record the data observed.
7. Analyze the data obtained.

34. Write down the properties of asymptotic notations?(MAY 2015) (R)
If $t1(n)\varepsilon\ O(g1(n))$ and $t2(n)\varepsilon\ O(g2(n))$, then
$t1(n)+t2(n)\varepsilon\ O(\max\{g1(n),g2(n)\})$

35. Give the Euclid algorithm for computing gcd(m, n) (MAY/JUN 2016) (Apr/May 17) (APR 17) (C)
ALGORITHM *Euclid_gcd(m, n)*
//Computes gcd*(m, n)* by Euclid's algorithm
//Input: Two nonnegative, not-both-zero integers *m* and *n*
//Output: Greatest common divisor of *m* and *n*
while $n \neq 0$ **do**
$r \leftarrow m$ mod *n*
m←n
n←r
return *m*
Example: gcd*(60, 24) =* gcd*(24, 12) =* gcd*(12, 0) =* 12.

36. Design an algorithm for computing area and circumference of the circle. (NOV/DEC 2016) (C)
//Computes area and circumference of the circle
//Input: One non-negative integer radius
//Output: Area and Circumference of the circle

```
Area = PI * rad * rad;
Ci = 2 * PI * rad;
return area, ci
```

37. How to measure an algorithm running time? (NOV/DEC 2017) (R)

One possible approach is to count the number of times each of the algorithm's operation is executed. The thing to do is identify the most important operation of the algorithm called as basic operation and compute the number of times the basic operation is executed.

$$T(n) \cong Cop\ C(n)$$

38. What is a basic operation? (APR/MAY 2018) (R)

The process of identify the most important operation of the algorithm called as basic operation

39. Define best,worst,average case time complexity. (NOV/DEC 2018)

Best case: In the best case analysis, we calculate lower bound on running time of an algorithm. We must know the case that causes minimum number of operations to be executed.

Worst case: In the worst case analysis, we calculate upper bound on running time of an algorithm. We must know the case that causes maximum number of operations to be executed.

Averge case: In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs.

40. How do you measure the efficiency of an algorithm. (APR/MAY 2019)

Time **efficiency** - a **measure** of amount of time for an **algorithm** to execute.

Space **efficiency** - a **measure** of the amount of memory needed for an **algorithm** to execute.

41. Prove that the if f(n)=O(g(n)) and g(n)=O(f(n)),then f(n)= θg (n) (APR/MAY 2019)

42. What do you mean bt interpolation search?

Interpolation search finds a particular item by computing the probe position. Initially, the probe position is the position of the middle most item of the collection.

If a match occurs, then the index of the item is returned. To split the list into two parts, we use the following method −

mid = Lo + ((Hi - Lo) / (A[Hi] - A[Lo])) * (X - A[Lo])

where −
  A   = list
  Lo = Lowest index of the list
  Hi = Highest index of the list

A[n] = Value stored at index n in the list

43. Define naïve string matching alogorithm.

The naïve approach tests all the possible placement of Pattern P [1…... m] relative to text T [1......n]. We try shift s = 0, 1.......n-m, successively and for each shift s. Compare T [s+1.......s+m] to P [1 .....m].

The naïve algorithm finds all valid shifts using a loop that checks the condition P [1.......m] = T [s+1…..... s+m] for each of the n - m +1 possible value of s.

NAIVE-STRING-MATCHER (T, P)

```
1. n ← length [T]
2. m ← length [P]
3. for s ← 0 to n -m
4. do if P [1.....m] = T [s + 1....s + m]
5. then print "Pattern occurs with shift" s
```

44. Define Rabin-Karp algorithm.

The Rabin-Karp string matching algorithm calculates a hash value for the pattern, as well as for each M-character subsequences of text to be compared. If the hash values are unequal, the algorithm will determine the hash value for next M-character sequence. If the hash values are equal, the algorithm will analyze the pattern and the M-character sequence. In this way, there is only one comparison per text subsequence, and character matching is onlyrequired when the hash values match.

RABIN-KARP-MATCHER (T, P, d, q)

1. $n \leftarrow$ length [T]
2. $m \leftarrow$ length [P]
3. $h \leftarrow d^{m-1}$ mod q
4. $p \leftarrow 0$
5. $t_0 \leftarrow 0$
6. for $i \leftarrow 1$ to m
7. do $p \leftarrow (dp + P[i])$ mod q
8. $t_0 \leftarrow (dt_0 + T[i])$ mod q
9. for $s \leftarrow 0$ to n-m
10. do if $p = t_s$
11. then if P [1.....m] = T [s+1. ... s + m]
12. then "Pattern occurs with shift" s
13. If $s < n$-m
14. then $t_{s+1} \leftarrow (d (t_s - T [s+1]h) + T [s+m+1])$ mod q

45. Define Knuth-Morris-Pratt algorithm.

Knuth-Morris and Pratt introduce a linear time algorithm for the string matching problem. A matching time of O (n) is achieved by avoiding comparison with an element of 'S' that have previously been involved in comparison with some element of the pattern 'p' to be matched. i.e., backtracking on the string 'S' never occurs

46. Write the advantage of insertion sort? (NOV/DEC 2017)

1. Simple implementation
2. Efficient for small data sets
3. Adaptive
4. More efficient in practice than most other simple quadratic, i.e. O(n2) algorithms such as Selection sort or bubble sort; the best case (nearly sorted input) is O(n)
5. Stable - does not change the relative order of elements with equal keys

47. Define Heap sort?

Heap Sort is one of the best sorting methods being in-place and with no quadratic worst-case running time. Heap sort involves building a **Heap** data structure from the given array and then utilizing the Heap to sort the array.
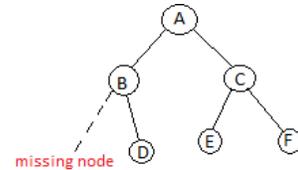
48. What are the properties of heap structure?

The special heap properties given below:

Shape Property: Heap data structure is always a Complete Binary Tree, which means all
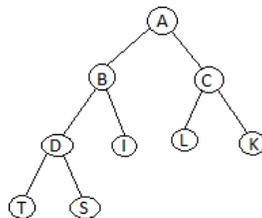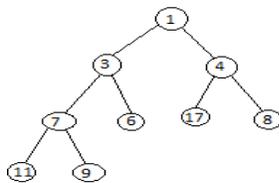

Complete Binary Tree   In-Complete Binary Tree

levels of the tree are fully filled.

Heap Property: All nodes are either **greater than or equal to** or **less than or equal to** each of its children. If the parent nodes are greater than their child nodes, heap is called a **Max-Heap**, and if the parent nodes are smaller than their child nodes, heap is called **Min-Heap**.

Min-Heap

In min-heap, first element is the smallest. So when we want to sort a list in ascending order, we create a Min-heap from that list, and picks the first element, as it is the smallest, then we repeat the process with remaining elements.
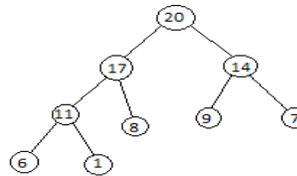
Max-Heap

In max-heap, the first element is the largest, hence it is used when we need to sort a list in descending order.

49. **What are the two basic parts of Heap sort ?**
- Creating a Heap of the unsorted list/array.
- Then a sorted array is created by repeatedly removing the largest/smallest element from the heap, and inserting it into the array. The heap is reconstructed after each removal.

50. What is the Complexity Analysis of Heap Sort?

Worst Case Time Complexity: **O(n*log n)**
Best Case Time Complexity: **O(n*log n)**
Average Time Complexity: **O(n*log n)**
Space Complexity : **O(1)**

51. What are the advantages of Heap sort ?

Heap sort is not a Stable sort, and requires a constant space for sorting a list.
Heap Sort is very fast and is widely used for sorting

PART B

1. Define the asymptotic notations used for best case average case and worst case analysis? **(APIRL/MAY2009) (APRIL/MAY-2008)(R)** **(Apr 18)**
2. How do you evaluate the performance of the algorithms? **(E)**
3. Explain properties of BIG (oh) Notation.(8) **(MAY 2016) (R) (Nov 17)**
4. What is meant by recurrence? Give one example to solve recurrence equations. **(APRIL/MAY 2012) (r)**
5. (i) Distinguish between Big Oh, Theta and Omega natation. **(NOV/DEC 2012) (AN)**
   (ii) Analyze the best, worst and average case analysis for linear search.

6. Find complexity of algorithm C (n) of the algorithm for the best, worst, average case,(evaluate average case complexity of n=3 n mean number of inputs.) **(E)**
7. (i) Define Asymptotic notations. Distinguish between Asymptotic notation and Conditional asymptotic notation. (10)**(APRIL/MAY 2010)(APRIL/MAY 2011) (Apr/May   17)**
   ii) Explain how the removing condition is done from the conditional asymptotic notation with an example. (6)**(NOV/DEC 2011) (R)**

8. (i) Explain how analysis of linear search is done with a suitable illustration. (10)
   (ii) Define recurrence equation and explain how solving recurrence equations are done.(6) **(NOV/DEC 2011) (R)**

9. Explain how Time Complexity is calculated. Give an example. **(APRIL/MAY 2010) (E)**

10. Discuss all the asymptotic notations in detail. **(APRIL/MAY 2012)(R)**

11. Write an algorithm for finding maximum element of an array, perform best, worst and average case complexity with appropriate order notations. **(APRIL/MAY 2008) (R)**

12. Write an algorithm to find mean and variance of an array perform best, worst and average case complexity, defining the notations used for each type of analysis. **(APRIL/MAY 2008) (AN)**

13. (i)Briefly explain the time complexity, space complexity estimation.(6) **(MAY/JUNE 2013)**

    (ii) Write the linear search algorithm and analyze its time complexity.(10) **(Nov/Dec 2016)**

14. (i)Find the time complexity and space complexity of the following problems. Factorial using recursion and Compute n^{th} Fibonacci number using Iterative statements. **(C)**

    (ii) Solve the following recurrence relations: **(NOV/DEC 2012)** **(A)**

$$1).T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + 3n & n > 2 \\ 2 & n = 2 \end{cases}$$

$$2)T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + cn & n > 1 \\ a & n = 1 \end{cases} \text{Where a and c are constants.}$$

15. Solve the following inequalities are correct **(MAY/JUNE 2013) (A)**

    (i)$5n^2 - 6n = \emptyset(n^2)$
    (ii) $n!=O(n^n)$
    (iii)$n^3 + 10n^2 = \theta(n^3)$
    (iv)$2n^2 2^n + n\log n = \theta(n^2 2^n)$

16. If you have to solve the searching problem for a list of n numbers, how can you take advantage of the fact that the list is known to be sorted? Give separate answer for

    (i) List represented as arrays
    (ii) List represented as linked lists
    Compare the time complexities involved in the analysis of both the algorithms.**(MAY 2015) (AN)**

17. (i) Derive the worst case analysis of merge sort using suitable illustrations.(8)**(MAY 2015)**
    (ii) Derive a loose bound on the following equation (8) **(A)**
       $F(x) =35x^8 -22x^7 +14x^5 - 2x^4-4x^2+x-15$

18. Give an algorithm to check whether all the Elements in a given array of n elements are distinct. Find the worst case complexity of the same.(8) **(MAY / JUNE 2016) (A)**

19. Give the recursive algorithm which finds the number of binary digits in the binary representation of a positive decimal integer. Find the recurrence relation and complexity. **(MAY / JUNE 2016) (A)**

20. State the general plan for analyzing the time efficiency of non-recursive algorithm and explain with an example. (8) **(Nov/Dec 2016) (AN)**

21. Solve the following recurrence relation. **(A)**
   - $x(n)=x(n-1)+5$ for $n>1$ $x(1)=0$
   - $x(n)=3x(n-1)$ for $n>1$ $x(1)=4$
   - $x(n)=x(n-1)+n$ for $n>1$ $x(0)=0$
   - $x(n)=x(n/2)+n$ for $n>1$ $x(1)=1$ (solve for $n=2^k$)
   - $x(n)=x(n/3)+1$ for $n>1$ $x(1)=1$ (solve for $n=3^k$) (16) **(Nov/Dec 2016)**

22. Briefly explain the mathematical analysis of recursive and non-recursive algorithm. (8) **(Apr/May 2017) (R)**

23. Discuss the steps in mathematical analysis for recursive algorithms. Do the same for finding the factorial of the number. **(NOV/DEC 2017)**

24. Give the general plan for analyzing the time efficiency of recursive algorithm and use recurrence to find number of moves Towers of Hanoi problem. **(APR/MAY 18)**

25. Consider the problem of finding the smallest and largest elements in an array of n numbers. **(APR/MAY 18)**
   (i) Design a presorting- based algorithm for solving this problem and determine its efficiency class. (7)

   (ii) Compare the efficiency of the three algorithms: (8)

 (A) the brute force algorithm (B) this presorting based algorithm (C) the divide and conquer algorithm.

26. (i)Prove that if g(n) is $\Omega(f(n))$ then f(n) is O(g(n)). (5) **(NOV/DEC 2018)**
   (ii) Discuss vrious methods used for mathematical analysis of recursive algorithms.(8) **(NOV/DEC 2018)**

27. Write the asymptotic notations used for best case,average case and worst case analysi of lgorithms. Writa an algorithm for finding maximum elements in an array. Give best,worst and average case complexities. **(NOV/DEC 2018)**

28. Solve the following recurrence relation: **(APR/MAY 2019)**
   (1) $T(n)=T(n/2)+1$, where $n=2^k$ for all $k \geq 0$ (4)

   (2) $T(n)=T(n/3)+T(2n/3)+cn$, where 'c' is a constant and 'n' is the input size.

29. Explains the steps involved in problem solving. **(APR/MAY 2019)**
30. What do you mean bt interpolation search?

31. Explain in detail about naïve string matching alogorithm.
32. Explain in detail about Rabin-Karp algorithm.
33. Explain in detail about Knuth-Morris-Pratt algorithm.
34. Explain in detail about insertion sort? (NOV/DEC 2017)


UNIT II

GRAPH ALGORITHMS

Graph algorithms: Representations of graphs - Graph traversal: DFS – BFS - applications - Connectivity, strong connectivity, bi-connectivity - Minimum spanning tree: Kruskal's and Prim's algorithm- Shortest path: Bellman-Ford algorithm - Dijkstra's algorithm - Floyd-Warshall algorithm Network flow: Flow networks - Ford-Fulkerson method – Matching: Maximum bipartite matching

1. **Define a graph. (April/May 2007) - U**

   A graph G=(V, E) consists of a set of vertices(V) and set of edges(E).

   In general, A *graph* is a collection of *nodes* (also called *vertices*) and *edges* (also called *arcs* or *links*) each connecting a pair of *nodes*.
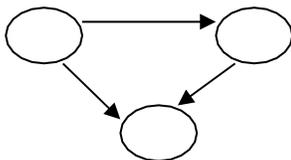
   Example:
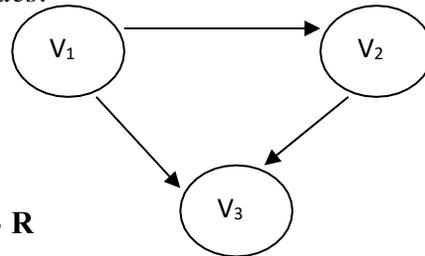
2. **What are the different types of Graph? - R**

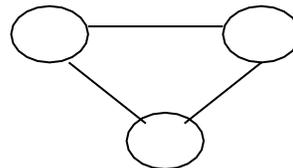   There are two types of Graphs. They are:

1. Directed Graphs.
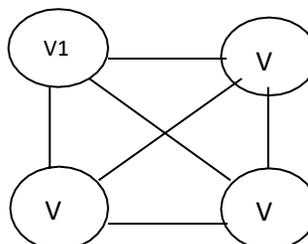2. Undirected Graphs.

   Example for Directed Graph: **Example for undirected Graph:**

3. **What is complete graph? - U**

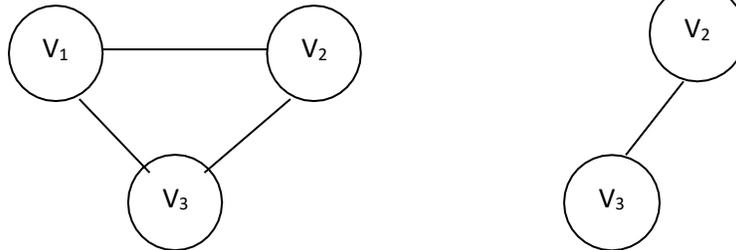   If an undirected graph of n vertices consists of n(n-1)/2 number of edges it is called as complete graph.

   Example:

4. What is sub graph? - U

A subgraph G' of graph G is a graph such that the set of vertices and set of edges of G' are proper subset of the set of edges of G.
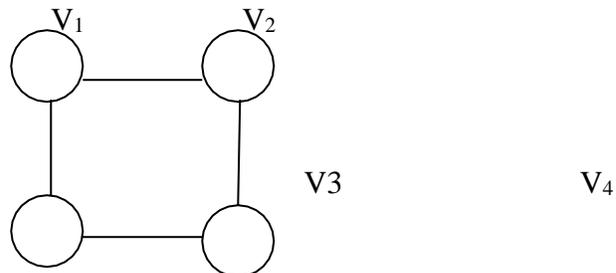
Example:

G Grapth   **G' Sub Graph**

5. What is connected graph? - U

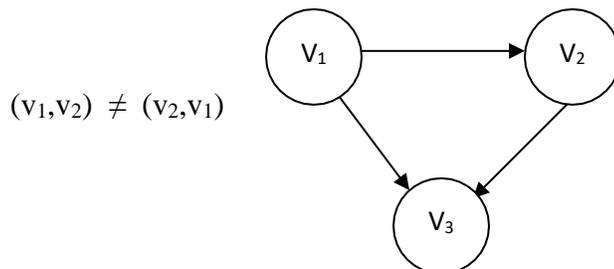An directed graph is said to be connected if for every pair of distinct vertices V_i and V_j in V(G) there is a graph from V_i to Vj in G.
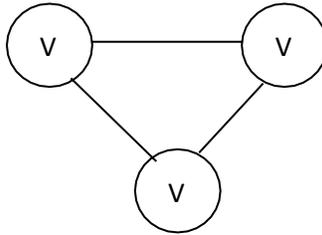
Example:

6. What are directed graphs? - U

Directed graph is a graph which consists of directed edges, where each edge in E is unidirectional. It is also referred as Digraph. If (v,w) is a directed edge then (v,w) $\neq$ (w,v

$(v_1,v_2) \neq (v_2,v_1)$

7. What are undirected graphs? - U

An undirected graph is a graph, which consists of undirected edges. If (v,w) is an undirected edge then (v,w) = (w,v)

$(v_1, v_2) = (v_2, v_1)$

8. Define cycle. - U

A cycle in a graph is a path in which first and last vertex are the same.

Path is 1 -> 2 -> 3 -> 1
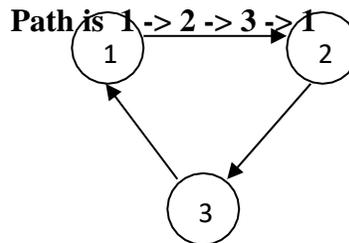
A graph which has a cycle is referred to as cyclic graph.

9. Define Acyclic graph. - U

A directed graph is said to be acyclic when there is no cycle path in it. It is also called DAG(Directed Acyclic Graph).

Example Path: A -> C -> D

10. Define weighted graph. - U

A graph is said to be weighted graph if every edge in the graph is assigned a weight or value. It can be directed or undirected graph.

Ex:

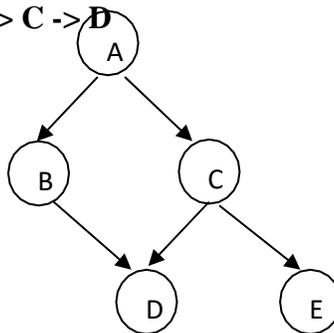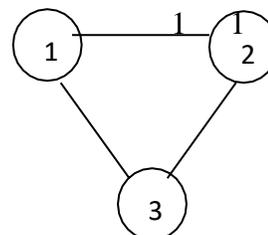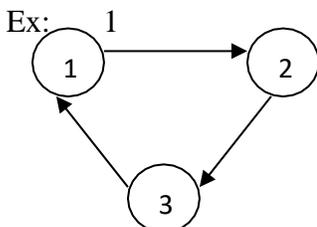## 11. Define strongly connected graph. - U

A directed graph is said to be strongly connected, if for every pair of vertex, there exists a path

Diagram:

V_1 V_2

V_4 V_3

V_5
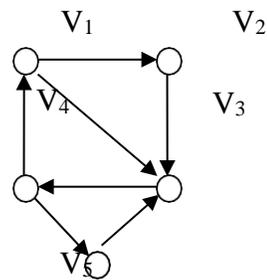
12. What is weakly connected graph.   (May/June 2012) - U

A digraph is weakly connected if all the vertices are connected to each other.

Diagram:

V_5 V_1 V_2

V_4 V_3
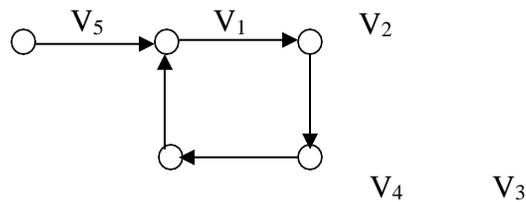
13. Define path in a graph. - U

A **path** in a graph is a sequence of vertices such that from each of its vertices there is an edge to the next vertex in the sequence. A path may be infinite, but a finite path always has a first vertex, called its *start vertex*, and a last vertex, called its *end vertex*. Both of them are called *end or terminal vertices* of the path. The other vertices in the path are *internal vertices*. A **cycle** is a path such that the start vertex and end vertex are the same.

14. What is the degree of a graph? - U

The number of edges incident on a vertex determines its degree. The degree of the vertex V is written as degree(V).

Example:

A   C

B   D
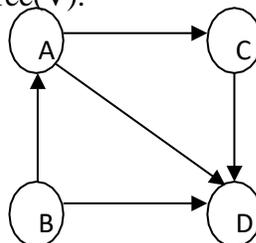
Degree (A) = 0

Degree (C) = 1

Degree (D) = 3

15. Define indegree, outdegree in a graph.   (April/May 2010) - U

The **indegree** is the numbers of edges entering in to the vertex V.

The **out degree** is the numbers of edges that are exiting or leaving from the vertex V.

Indegree(V₁) = 1

Outdegree (V₁) = 2
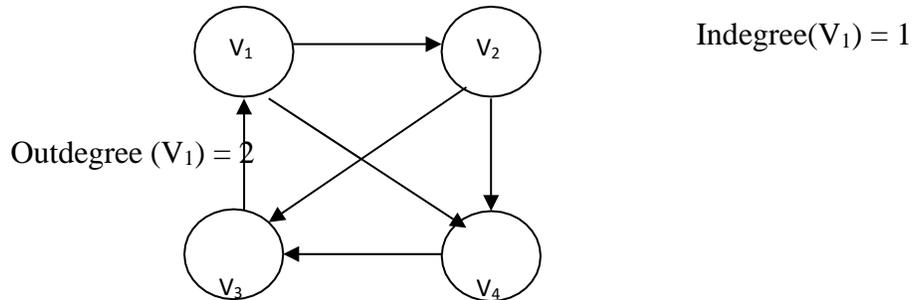
16. What is adjacent node? - U

Adjacency node is the node which is connected to the other nodes in the graph.

Example:
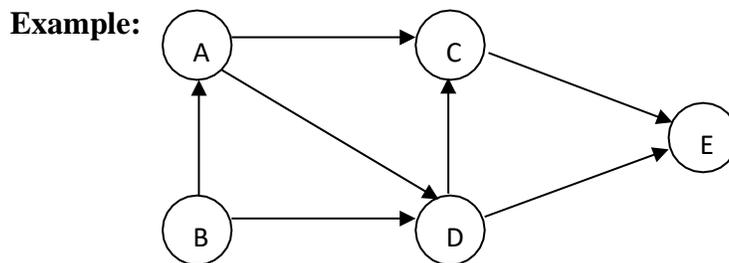
In the above diagram, C is an adjacent node to E

17.   What are the applications of graphs? - R

The various applications of graphs are:

- In computer networking such LAN, WAN etc., internetworking of computer systems is done using graph concept.
- In telephone cabling, graph theory is effectively used.
- In job scheduling algorithms, the graph is used.
  Solution: a b c d

18. Prove that the number of odd degree vertices in a connected graph should be even. (May/June 2007) - An

Consider a graph with odd degree vertices.
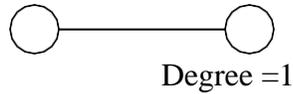
Degree =1

No. of vertices = 2

Degree = 3
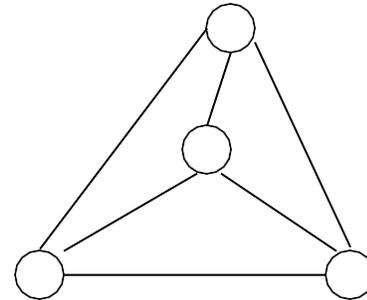
No. of vertices = 4

This implies that the no. of vertices is even for odd degree graphs.

19. What is a single source shortest path problem? - U

The single source shortest path problem finds the minimum cost from single source vertex to all other vertices. Dijkstra's algorithm is used to solve this problem which follows the greedy technique.

20. What is unweighted shortest path? -- U

The unweighted shortest path is the path in unweighted graph which is equal to number of edges traveled from source to destination.

Example:
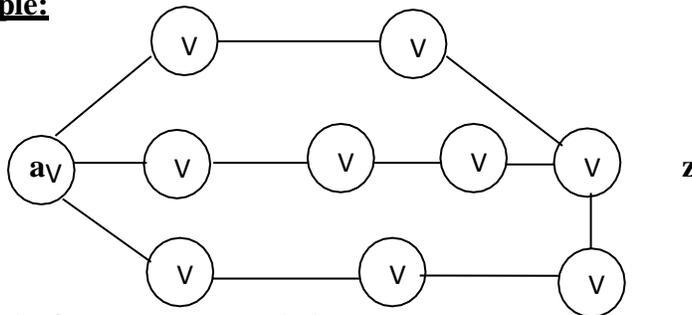
The paths from **a to z** are as below:

S.No	Path	Number of edges
1	V_1-V_2-V_3-	3
2	V_1-V_4-V_5-	4
3	V_1-V_7-V_8-	4

V_1-V_2-V_3-V_{10} is the shortest path.

21. What are the different kinds of graph traversals? -R

- Depth-first traversal
- Breadth-first traversal

22. What is depth-first traversal? Give an example. -- U

   Depth-first search (**DFS**) is a graph search algorithm that begins at the root and explores as far as possible along each branch before backtracking.
Note: Backtracking is finding a solution by trying one of several choices. If the choice proves incorrect, computation *backtracks* or restarts at the point of choice and tries another choice.

   Example:

   The Depth-first traversal is **1 – 2 – 4 – 3**
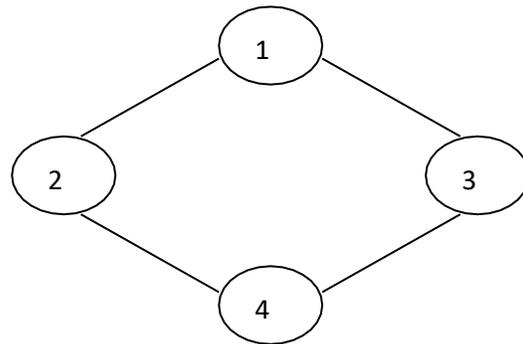
23. What is breadth-first traversal? Give an example. - U

   Breadth-first search (**BFS**) is a graph search algorithm that begins at the root node and explores all the neighboring nodes. Then for each of those nearest nodes, it explores their unexplored neighbor nodes, and so on, until it finds the goal.

   Example:
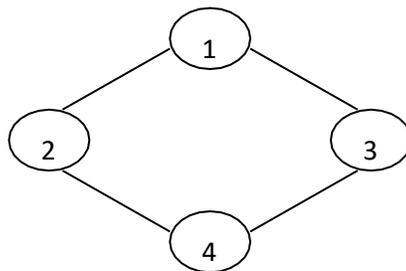
   The breadth-first Traversal is **1 – 2 – 3 – 4**

24. What is biconnectivity? – U

Biconnected graphs are the graphs which cannot be broken into two disoconnected graphs by disconnecting single edge.

Example
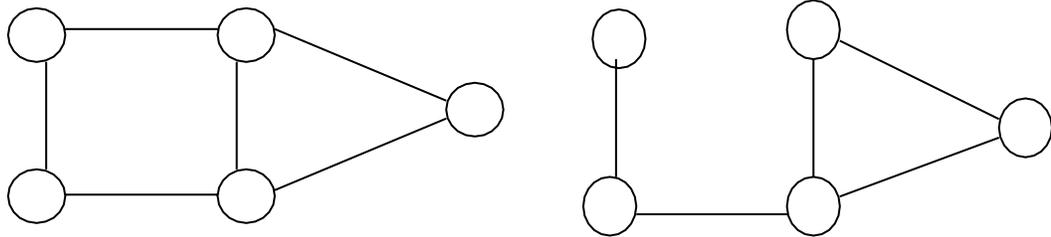
In the given example, after removing edge E_1 the graph does not become disconnected.

25. What is a Spanning Tree?

Given an undirected and connected graph G=(V,E), a spanning tree of the graph G is a tree that spans G (that is, it includes every vertex of G) and is a subgraph of G (every edge in the tree belongs to G)

26. Define prims algorithm.

Prim's Algorithm is a greedy algorithm that is used to find the minimum spanning tree from a graph. Prim's algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.

Prim's algorithm starts with the single node and explores all the adjacent nodes with all the connecting edges at every step. The edges with the minimal weights causing no cyclesin the graph got selected.

27. Define Kruskal algorithm.

Kruskal's Algorithm is used to find the minimum spanning tree for a connected weighted graph. The main target of the algorithm is to find the subset of edges by using which we can traverse every vertex of the graph. It follows the greedy approach that findsan optimum solution at every stage instead of focusing on a global optimum.

28. Define Bellman Ford algorithm.

Bellman Ford algorithm helps us find the shortest path from a vertex to all other vertices of a weighted graph.It can work with graphs in which edges can have negative weights.

29. Define Dijkstra's algorithm.

Dijkstra's Algorithm finds the shortest path between a given node (which is called the "source node") and all other nodes in a graph.This algorithm uses the weights of the edges to find the path that minimizes the total distance (weight) between the source node and all other nodes.

30. Define Floyd-Warshall algorithm.

The all pair shortest path algorithm is also known as Floyd-Warshall algorithm is used to find all pair shortest path problem from a given weighted graph. As a result of this algorithm, it will generate a matrix, which will represent the minimum distance from any node to all other nodes in the graph.
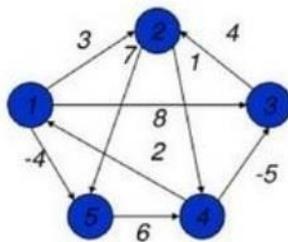
$$\begin{vmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{vmatrix}$$

At first the output matrix is same as given cost matrix of the graph. After that the output matrix will be updated with all vertices k as the intermediate vertex.

31. Give the Floyd's algorithm (C) (Nov 17)

```
ALGORITHM Floyd(W[1..n,1..n])
//Implements Floyd's algorithm for the all-pair shortest–path problem
//Input The weight matrix W of a graph
//Output The distance matrix of the shortest paths' lengths
D <- W
for k = 1 to n do
for i =1 to n do
for j =1 to n do
D[I,j] = min{D[I,j], D[I,k] + D[k,j]
```

32. Define Prims Algorithm. (R)

Prim's algorithm constructs a minimum spanning tree through a sequenceof expanding subtrees. The initial subtree in such a sequence consists of a singlevertex selected arbitrarily from the set V of the graph's vertices. On each iteration,the algorithm expands the current tree in the greedy manner by simply attaching toit the nearest vertex not in that tree.

33. Write down the optimization technique used for Warshalls algorithm. State the rules and assumptions which are implied behind that. (MAY 2015) (R)

Warshalls algorithm constructs the transitive closure of given digraph with n vertices through a series of n-by-n Boolean matrices. The computations in warshalls algorithm are given by following sequences,

$R^{(0)},\ldots,R^{(k-1)},\ldots,R^{(k)},\ldots,R^{(n)}$

Rules:

1.Start with computation of $R^{(0)}$. In $R^{(0)}$ any path with intermediate veritices is not allowed. That means only direct edges towards the vertices are considered. In other wordsthe path length of one edge is allowed in $R^{(0)}$. Thus $R^{(0)}$ is adjacency matrix for the digraph.

2. Construct $R^{(1)}$ in which first vertex is used as intermediate vertex and a path length of two edge is allowed. Note that $R^{(1)}$ is build using $R^{(0)}$which is already computed.

3. Go on building $R^{(k)}$ by adding one intermediate vertex each time and with more path length. Each $R^{(k)}$ has to be built from $R^{(k-1)}$

4. The last matrix in this series is $R^{(1)}$, in this $R^{(n)}$ all the n vertices are used as intermediate vertices. And the $R^{(n)}$ which is obtained is nothing but the transitive closure of given digraph.

34. Define the single source shortest path problem. (MAY\JUNE 2016) (R)

Dijkstra's algorithm solves the single source shortest path problem of finding shortest paths from a given vertex(the source), to all the other vertices of a weighted graph or digraph.

Dijkstra's algorithm provides a correct solution for a graph with non-negative weights.

35. How to calculate the efficiency of dijkstra's algorithm. (NOV/DEC 16) (R)

The time efficiency depends on the data structure used for implementing the priority queue and for representing the input graph. It is in θ $(|V|)^2$ for graphs represented by their weight matrix and the priority queue implemented as an unordered array. For graphs represented by their adjacency linked lists and the priority queue implemented as a min heap it is in $O(|E|\log|V|)$.

36. Define transitive closure of a directed graph. (APR/MAY 2018)

The transitive closure of a directed graph with n vertices can be defined as the "n-by-n" Boolean matrix T={tij} in which,the element in the ith row $(1<i<n)$ and the jth column $(1<j<n)$ exists a non-trival directed path from the Ith vertex to the jth vertex,otherwise tij is 0.

37. Write short notes on the Maximum-Flow problem. (R)

Maximum-flow algorithms require processing edges in both directions, it is convenient to modify the adjacency matrix representation of a network as follows. If there is a directed edge from vertex i to vertex j of capacity uij ,then the element in the ithrow and the jth column is set to uij , and the element in the jth row and the ith column is

set to −uij; if there is no edge between vertices i and j , both these elements are set to zero. Outline a simple algorithm for identifying a source and a sink in a network presented by sucha matrix and indicate its time efficiency.

38. Define maximum matching. (R)

In many situations we are faced with a problem of pairing elements of two sets.The traditional example is boys and girls for a dance, but you can easily thinkof moreserious applications. It is convenient to represent elements of two givensets by vertices ofa graph, with edges between vertices that can be paired. Amatching in a graph is a subset of its edges with the property that no two edgesshare a vertex. A maximum matching— more precisely, a maximum cardinalitymatching—is a matching with the largest number of edges.

39. Define maximum cardinality. (R) (NOV/DEC 2016) (R) (Apr 18)

Maximum cardinality matching—is a matching with the largest number of edges. The maximum-matching problem is the problem offinding a maximum matching in a given graph. For an arbitrary graph, this is arather difficult problem. It was solved in 1965 by Jack Edmonds.

40. Define maximum flow problem. (R)

The maximum flow problem can be seen as a special case of more complex network flow problems, such as the circulation problem. The maximum value of an s-t flow (i.e., flow from source s to sink t) is equal to the minimum capacity of an s-t cut (i.e., cut severing s from t) in the network, as stated in the max-flow min-cut theorem.

41. Define Bipartite Graphs? (R) (Nov 17)

A bipartite graph, also called a bigraph, is a set of graph vertices decomposed into two disjoint sets such that no two graph vertices within the same set are adjacent. A bipartite graph is a special case of a k-partite graphwith $k = 2$. The illustration above shows some bipartite graphs, with vertices in each graph colored based on to which of the two disjoint sets they belong.

42. What do you meant by perfect matching in bipartite graph? (APR/MAY 2017) (R)

When there are an equal number of nodes on each side of a bipartite graph, a perfect matching is an assignment of nodes on the left to nodes on the right, in such a waythat
   i)each node is connected by an edge to the node it is assigned to, and
   ii)no two nodes on the left are assigned to the same node on the right

43. Define flow cut. (R)

       Cut is a collection of arcs such that if they are removed there is no path from s to t. A cut is said to be minimum in a network whose capacity is minimum over all cuts of the network.

44. How is a transportation network represented? (R)(APR/MAY 18)

    The transportation network can be represented by a connected weighted digraph with n vertices numbered from 1 to n and a set of edges E, with the following properties:

- It contains only one vertex with no entering edges called as **source** and assumed to be 1.
- It contains only one vertex at end with no leaving edges called as **sink** and assumed as n.
- The weight u_{ij} of each directed edge (i, j) is a positive integer, called as **edge capacity**.

45. Define the constraint in the context of maximum flow problem. (APR/MAY 2019)

It represents the maximum amount of flow that can pass through an edge. ... , for each (capacity constraint: the flow of an edge cannot exceed its capacity); , for each (conservation of flows: the sum of the flows entering a node must equal the sum of the flows exiting a node, except for the source and the sink nodes).

PART –B

1. Write an algorithm for all pairs shortest path algorithm and what are the time and space complexity of the algorithm. **(APIRL/MAY2009) (APRIL/MAY 2012) (R)**

2. Explain Floyd algorithm with example. Write down and explain the algorithm to solve all pairs shortest paths problem. **(APRIL/MAY 2010)(MAY/JUNE 2013). (R)**
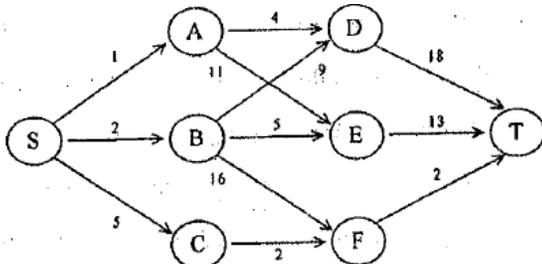
3. With a suitable example, explain all-pair shortest paths problem. **(R)**

4. How do you construct a minimum spanning tree using Kruskals algorithm? Explain?(4) **(MAY 2015) (R)**

5. Discuss about the algorithm and pseudocode to find the Minimum Spanning Tree using Prim's Algorithm. Find the Minimum Spanning Tree for the graph. Discuss about the efficiency of the algorithm**.(MAY\JUNE 2016) (C) (Apr 18)**

   6. Solve the all-Pairs shortest-path problem for the diagraph with the following weight

matrix:
$$\begin{vmatrix} 0 & 2 & \infty & 1 & 8 \\ 6 & 0 & 3 & 2 & \infty \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 3 & \infty & \infty & \infty & 0 \end{vmatrix}$$
(NOV/DEC 2016) (A)

7. Apply Kruskal's algorithm to find a minimum spanning tree of the following graph.
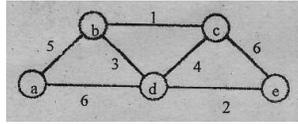
(NOV/DEC 2016) (A)

8. Explain the Dijkstra's shortest path algorithm and its efficiency.**(R) (NOV/DEC 2017)**

9. Write down the Dijkstra's algorithm and explain it with an example (8)

(APR/MAY 11) – Ap

10. Explain Dijkstra's algorithm with example.**(May/June 2012, Nov/Dec 2012) – Ap**

11. Write suitable ADT operation for shortest path problem. Show the simulation of shortest path with an example graph.   **(April/May 2008) – Ap**

12. What is single source shortest path problem? Discuss Dijkstra's single source shortest path

algorithm with an example. (8)   **(April/May 2007) –Ap**

13. Explain Depth – first & Breadth – First Traversal algorithms. – U

14. Explain depth first search on a graph with necessary data structures. (8)

(April/May 2008) – U

15. Write short notes on Biconnectivity? **– U**

16. Explain Dijkstra's algorithm using the following graph. Find the shortest path between v1, v2, v3, v4, v5, v6 & v7.   2

    (May/June 2007) - Ap
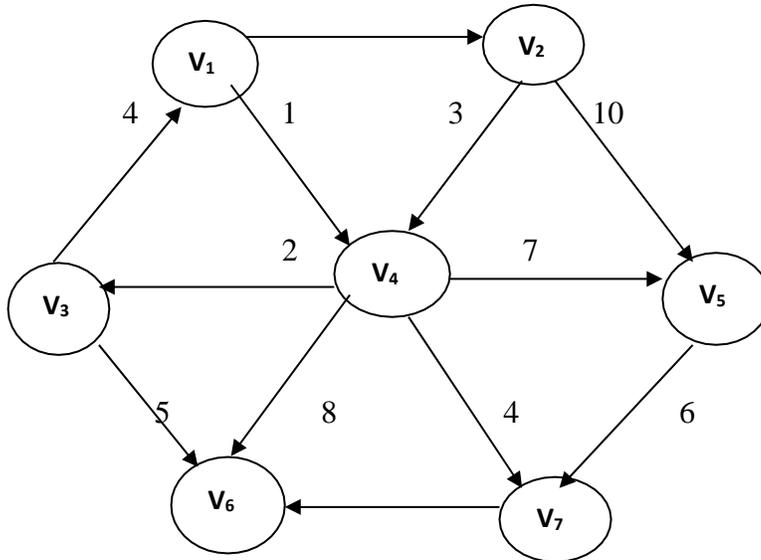
17. Distinguish between breadth first search and depth first search with example.**(13)**

    (NOV/DEC 2018)

18. Explain the algorithm for Maximum Bipartite Matching. **(R)**
19. How do you compute maximum flow for the following graph using Ford-Fulkerson method? **(MAY 2015) (E)**
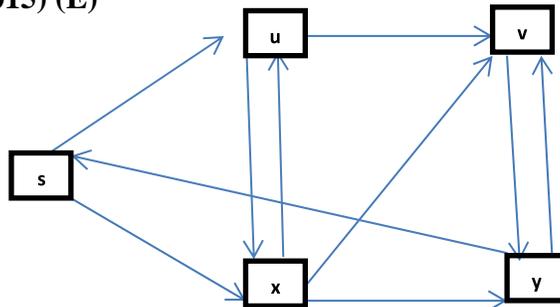
20. State and Prove Maximum Flow Min cut Theorem. **(MAY\JUNE 2016) (R)**
21. Apply the shortest Augmenting Path algorithm to the network shown below. **(MAY\JUNE 2016) (A)**
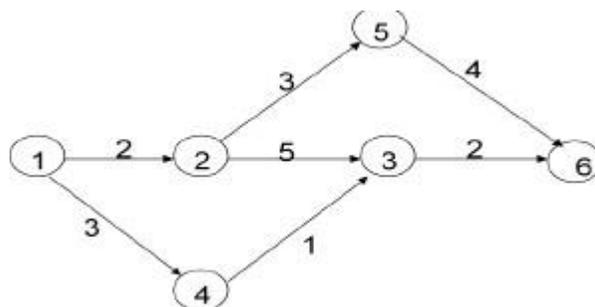
22. Explain   KMP   string

matching algorithm for finding a pattern on the text, a analysis the algorithm**.(APR/MAY 2017) (R)**

23. Determine the max-flow in the following network.(13) **(APR/MAY 2019)**
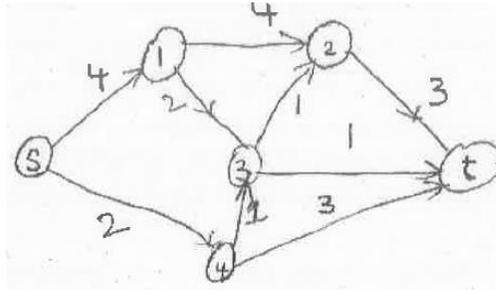
UNIT III

ALGORITHM DESIGN TECHNIQUES

Divide and Conquer methodology: Finding maximum and minimum - Merge sort - Quick sort **Dynamic programming:** Elements of dynamic programming — Matrix-chain multiplication - Multi stage graph — Optimal Binary Search Trees. **Greedy Technique:** Elements of the greedy strategy - Activity-selection problem –- Optimal Merge pattern — Huffman Trees.

PART A

1. **Give the general plan for divide-and-conquer algorithms. (APRIL/MAY 2008) (NOV/DEC 2008) (MAY/JUNE 2016) (R) (Nov 17)**
   The general plan is as follows.
   A problems instance is divided into several smaller instances of the same problem, ideally about the same size.
   The smaller instances are solved, typically recursively.
   If necessary the solutions obtained are combined to get the solution of the original problem.

2. **List the advantages of Divide and Conquer Algorithm.**
   Solving difficult problems, Algorithm efficiency, Parallelism, Memory access, Round off control.

3. **Give the recurrence relation of divide-and-conquer? (R)**
   The recurrence relation is

   $$T(n) = \begin{cases} g(n) \\ t(n1) + t(n2) + ... + t(nk) + f(n) \end{cases} \quad g(n)$$

4. **Define of feasibility.**

A feasible set (of candidates) is promising if it can be extended to produce not merely a solution, but an optimal solution to the problem.

5. **Define Quick Sort.**

Quick sort is an algorithm of choice in many situations because it is not difficult to implement, it is a good \"general purpose\" sort and it consumes relatively fewer resources during execution.

6. **List out the Disadvantages in Quick Sort**
   1. It is recursive. Especially if recursion is not available, the implementation is extremely complicated.
   2. It requires quadratic (i.e., n2) time in the worst-case.
   3. It is fragile i.e., a simple mistake in the implementation can go unnoticed and cause it to perform badly.

7. **What is the difference between quicksort and mergesort?**

Both quicksort and mergesort use the divide-and-conquer technique in which the given array is partitioned into subarrays and solved. The difference lies in the technique that the arrays are partitioned. For mergesort the arrays are partitioned according to their position and in quicksort they are partitioned according to the element values.

8. **Define merge sort. (R)**
   b. Mergesort sorts a given array A[0..n-1] by dividing it into two halves a[0..(n/2)-1] and A[n/2..n-1] sorting each of them recursively and then merging the twosmaller sorted arrays into a single sorted one.

9. **List the Steps in Merge Sort**

   Divide Step: If given array A has zero or one element, return S; it is already sorted. Otherwise, divide A into two arrays, A1 and A2, each containing about half of the elements of A.

   Recursion Step: Recursively sort array A1 and A2.

   Conquer Step: Combine the elements back in A by merging the sorted arrays A1 and A2 into a sorted sequence

10. **List out Disadvantages of Divide and Conquer Algorithm**

    Conceptual difficulty
    Recursion overhead
    Repeated subproblems

11. **List out the Advantages in Quick Sort**
    - It is in-place since it uses only a small auxiliary stack.
    - It requires only n log(n) time to sort n items.
    - It has an extremely short inner loop

This algorithm has been subjected to a thorough mathematical analysis, a very precise statement can be made about performance issues.

12. Describe the recurrence relation of merge sort? (R)

If the time for the merging operation is proportional to n, then the computing time of merge sort is described by the recurrence relation

$T(n)= a$

$2T(n/2)+n$ n = 1, a constant

13. State Master's theorem. (APR/MAY 18)

If $f(n)$ $\theta(n_d)$ where d ^{3} 0 in recurrence equation $T(n) = aT(n/b)+f(n)$, then

θ (n_d) if $a<b_d$ $T(n)$ θ $(n_d\log n)$ if $a=b_d$

θ $(n\log b_a)$ if $a>b_d$

The efficiency analysis of many divide-and-conquer algorithms is greatly simplified by the use of Master theorem.

14. List out the Disadvantages in Quick Sort

• It is recursive. Especially if recursion is not available, the implementation is extremely complicated.

• It requires quadratic (i.e., n2) time in the worst-case.

• It is fragile i.e., a simple mistake in the implementation can go unnoticed and cause it to perform badly.

15. What are the differences between dynamic programming and divide and conquer approaches? (NOV/DEC 2018)

Divide and Conquer

Divide and Conquer works by dividing the problem into sub-problems, conquer each sub-problem recursively and combine these solutions.

Dynamic Programming

Dynamic Programming is a technique for solving problems with overlapping subproblems. Each sub-problem is solved only once and the result of each sub-problem isstored in a table (generally implemented as an array or a hash table) for future references. These sub-solutions may be used to obtain the original solution and the technique of storing the sub-problem solutions is known as memoization.

16.   What is the time and space complexity of Merge sort? (APR/MAY 2019)

Time complexity = $\theta(n \log n)$

Space complexity = $n+\log_2 n$

$=\theta(n)$

17. Define dynamic programming. (APR/MAY 2017) (R)

Dynamic programming is an algorithm design method that can be used when asolution to the problem is viewed as the result of sequence of decisions.

18. What are the features of dynamic programming? (R)

➢ Optimal solutions to sub problems are retained so as to avoid re-computing their values.

➢ Decision sequences containing subsequences that are sub optimal are not considered.

➢ It definitely gives the optimal solution always.

19. What are the drawbacks of dynamic programming? (R)

➢ Time and space requirements are high, since storage is needed for all level.

➢ Optimality should be checked at all levels.

20. Write the general procedure of dynamic programming.(APR/MAY 2017) (R)

The development of dynamic programming algorithm can be broken into a

sequence of 4 steps.

1. Characterize the structure of an optimal solution.
2. Recursively defines the value of the optimal solution.
3. Compute the value of an optimal solution in the bottom-up fashion.
4. Construct an optimal solution from the computed information.

21. Write the difference between the Greedy method and Dynamic programming.(APRIL/MAY 2011)(NOV/DEC 2012) (AN)

1.Only one sequence of decision is generated.	1.Many number of decisions are
2.It does not guarantee to give an optimal solution always.	2.It definitely gives an optimal

22. Define the principle of optimality. (APR/MAY 2012) (NOV/DEC 2016) (R) (Nov 17)

It states that in an optimal sequence of decisions, each sub sequence must be optimal. Touse dynamic programming the problem must observe the principle of optimality thatwhatever the initial state is, remaining decisions must be optimal with regard the statefollowing from the first decision.

23. What is the difference between dynamic programming and greed algorithm?(APRIL/MAY 2012) (AN)

GREEDY ALGORITHM	DYNAMIC PROGRAMMING
1. GA computes the solution in bottom up technique. It computes the solution from smaller sub-routines and tries many possibilities and choices before	DA computes its solution by making its choices in a serial fashion (never look back irrevocable).

it arrives at the optimal set of choices.	
It does not use the principle of optimality, i.e. there is no test by which one can tell GA will lead to an optimal solution.	It uses the principle of optimality.

24. Define Optimal Binary Search Trees? (R)

A binary search tree is one of the most important data structures in computer science. One of its principal applications is to implement a dictionary, a set of elements with the operations of searching, insertion, and deletion. If probabilities of searching for elements of a set are known e.g., from accumulated data about past searches - it is natural to pose a question about an optimal binary search tree for which the average number of comparisons in a search is the smallest possible.

25. List out the memory functions used under Dynamic programming. (MAY 2015) (R)

Memory functions solve in a top-down manner only sub problems that are necessary. Memory functions are an improvement of dynamic programming because they only solve sub problems that are necessary and do it only once. However they require more because it makes recursive calls which require additional memory.

- Greedy Algorithm
- Branch and Bound
- Genetic Algorithm

26. State the general principle of greedy algorithm. (NOV/DEC 16) (R)

A greedy algorithm is an algorithmic paradigm that follows the problem solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum.

27. Define Optimization function ?

Every set of s that satisfies the constraints is a feasible solution.
Every feasible solution that maximizes is an optimal solution.

29. Define Greedy Methodology?

Greedy algorithms are simple and straightforward.
They are shortsighted in their approach
A greedy algorithm is similar to a dynamic programming algorithm, but the difference is that solutions to the sub problems do not have to be known at each stage, □ instead a "greedy" choice can be made of what looks best for the moment.

30. Define an optimization problem?

Given a problem instance, a set of constraints and an objective function.

Find a feasible solution for the given instance for which the objective function has an optimal value.
Either maximum or minimum depending on the problem being solved.A feasible solution that does this is called optimal solution.

31. Write all the Greedy Properties?
It consists of two property,
1. "greedy-choice property" ->It says that a globally optimal solution can be arrived at by making a locally optimal choice.
2. "optimal substructure" ->A problem exhibits optimal substructure if an optimal solution to the problem contains optimal solutions to the sub-problems.
are two ingredients in the problem that lend to a greedy strategy.

32. State feasible and consrtraint ?
Feasible: A feasible solution satisfies the problem's constraints
Constraints: The constraints specify the limitations on the required solutions

33. Write the Pseudo-code for Greedy Algorithm
Algorithm Greedy (a,n)
//a[1:n]contains the n inputs.
{
solution:=0;//initialize the solution.
for i:=1 to n do
{
x:=Select(a);
if Feasible(solution, x) then
solution:=Union(solution,x);
}
return solution

34. Compare With Dynamic Programming ?
Greedy And Dynamic Programming are methods for solving optimization problems.
Greedy algorithms are usually more efficient than DP solutions. However, often you need to use dynamic programming since the optimal solution cannot be guaranteed by a greedy algorithm.
DP provides efficient solutions for some problems for which a brute force approach would be very slow.

35. Differentiate PROS AND CONS
PROS:
- They are easier to implement,
- They require much less computing resources,
- They are much faster to execute.
- Greedy algorithms are used to solve optimization problems

CONS:
- Their only disadvantage being that they not always reach the global optimum solution;

- on the other hand, even when the global optimum solution is not reached, most of the times the reached sub-optimal solution is a very good solution

36. Comment on merge pattern?

Merge a set of sorted files of different length into a single sorted file. We need to find an optimal solution, where the resultant file will be generated in minimum time.

If the number of sorted files are given, there are many ways to merge them into a single sorted file. This merge can be performed pair wise. Hence, this type of merging is called as **2-way merge patterns**.

37. Define multistage graphs. Give an example. (NOV/DEC 2018)

A multistage graph $G = (V, E)$ is a directed graph where vertices are partitioned into k (where $k > 1$) number of disjoint subsets $S = \{s1,s2,…,sk\}$ such that edge (u, v) is in E, then $u \in si$ and $v \in s1 + 1$ for some subsets in the partition and $|s1| = |sk| = 1$.

The vertex $s \in s1$ is called the source and the vertex $t \in sk$ is called sink.
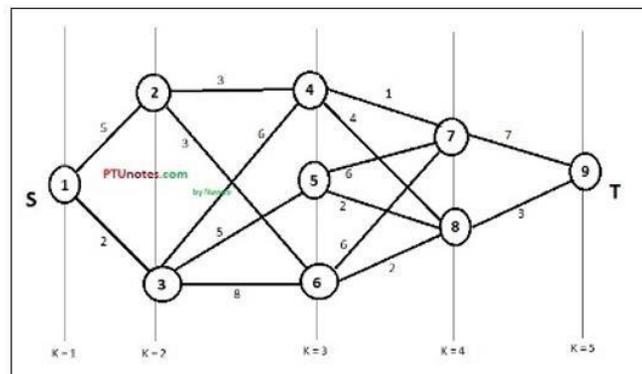
38. State the principle of optimality. (APR/MAY 2019)

In an optimal sequence of decisions or choices, each subsequence must also be optimal. When it is not possible to apply the principle of optimality it is almost impossible to obtain the solution using the dynamic programming approach.
Example: Finding of shortest path in a given graph uses the principle of optimality.

PART-B

1. Define divide and conquer to apply the technique in binary search algorithm and to analysis it. **(APR/MAY 2006) (APR/MAY 2017) (R)**
2. Explain in detail in merge sort give an example **(APR/MAY 2008) (MAY 2016). (R)**
3. What is divide and conquer strategy and explain the binary search with suitable example problem.**(NOV/DEC 2011) (R)**

4. Distinguish between Quick sort and Merge sort, and arrange the following numbers in increasing order using merge sort. (18, 29, 68, 32, 43, 37, 87, 24, 47, 50) **(NOV/DEC 2011) (MAY/JUNE 2013). (A)**

5. Trace the steps of Mergesort algorithm for the elements 122,25,70,175,89,90,95,102,123 and also compute its time complexity.**(NOV/DEC 2012) (A) (Nov 17) (Apr 18)**

6. Write an algorithm to perform binary search on a sorted list of elements. Analyze the algorithm for the best case, average case and worst case.**(APR/MAY 2011). (AN)**

7. Using the divide and conquer approach to find the maximum and minimum in a set of 'n' elements. Also find the recurrence relation for the number of elements compared and solve the same.**(APR/MAY 2011). (A)**

8. Trace maximum and minimum (using divide and conquer) algorithm for the following set of numbers. 20, 35, 18, 8, 14, 41, 3, 39,-20. **(A)**

9. Write a pseudo code using divide and conquer technique for finding the position of the largest element in an array of N numbers. **(A)**

10. Sort the following set of elements using merge sort: 12, 24, 8, 71, 4, 23, 6, 89, and 56. **(A)**

11. Explain in detail quick sorting method. Provide a complete analysis of quick sort. **(APR/MAY 2008) (Nov/Dec 2016) (APR/MAY 2017) (AN)**

12. A pair contains two numbers and its second number is on the right side of the first one in an array. The difference of a pair is the minus result while subtracting the second number from the first one. Implement a function which gets the maximal difference of all pairs in an array (using divide and conquer method). **(MAY/JUNE 2015) ®**

13. Write the algorithm for quick sort. Provide a complete analysis of quick sort for the given set of numbers 12,33,23,43,44,55,64,77 and 76. (13)**(NOV/DEC 2018)**

14. Write the quick sort algorithm and explain it with an example. Derive the worst case and average case time complexity. (5+4+4) **(APR/MAY 2019)**

15. (i)Write an algorithm to construct the optimal binary search tree (or) Discuss the algorithm for finding a minimum cost binary search trees.(8)

(ii) Explain how dynamic programming is applied to solve travelling salesperson problem. **(APR/MAY 2010)(NOV/DEC 2012)(8) (R)**

16. Using Dynamic approach programming, solve the following graph using the backward approach.**(APRIL/MAY2011)** **(A)**
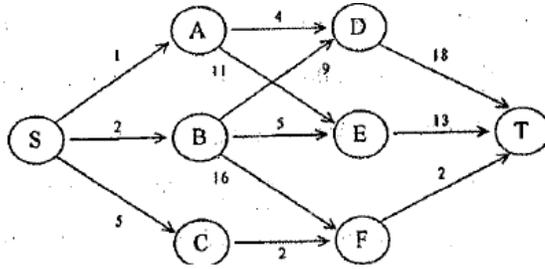
17. (i) Let A ={l/119,m/96,c/247,g/283,h/72,f/77,k/92,j/19} be the letters and its frequency of distribution in a text file. Compute a suitable Huffman coding to compress the data effectively. (8) **(MAY 2015)**

  (ii)Write an algorithm to construct the optimal binary search tree given the roots r(i,j) ,$0 \le i \le j \le n$. Also prove that this could be performed in time O(n). (8) **(MAY 2015) (AN)**

18. Write the Huffman's Algorithm. Construct the Huffman's tree for the following data and obtain its Huffman's Code. **(APR/AMY 2017) (A)**

Characters	A	B	C	D	E	_
Probability	0.5	0.35	0.5	0.1	0.4	0.2

19. Explain the steps in building a Huffman Tree. Find the codes for the alphabets given below as according to frequency **(NOV/DEC 2017)**

A	2
E	5
H	1
I	2
L	2
M	2
P	2
R	1
S	2
X	1

20. (i) Write the Huffman code algorithm and derive its time complexity. **(5+2) (APR/MAY 2019)**

(ii) Generate the Huffman code for the following data comprising of alphabet and their frequency.(6) (APR/MAY 2019)

a:1,b:1,c:2,d:3,e:5,f:8,g:13,h:21

UNIT IV
STATE SPACE SEARCH ALGORITHMS

Lower - Bound Arguments - P, NP NP- Complete and NP Hard Problems. Backtracking – n- Queen problem - Hamiltonian Circuit Problem – Subset Sum Problem. Branch and Bound – LIFO Search and FIFO search - Assignment problem – Knapsack Problem – Travelling Salesman Problem - Approximation Algorithms for NP-Hard Problems – Travelling Salesman problem – Knapsack problem.

PART - A

1. **Differentiate backtracking and Exhaustive search. (AN)**

S.No	Backtracking	Exhaustive search
1.	Backtracking is to build up the solution vector one component at a time and to use modified criterion Function Pi(x1,..,xi) (sometimes called bounding function) to test whether the vector being formed has any chance of success.	Exhaustive search is simply a "brute – force" approach to combinatorial problems. It suggests generating each and every element of the problem's domain, selecting those of them that satisfy the problem's constraints, and then finding a desired element.
2.	Backtracking makes it possible to solve many large instances of NP-hard problems in an acceptable amount of time.	Exhaustive search is impractical for large instances, however applicable for small instances of problems.

2. **What are the factors that influence the efficiency of the backtracking algorithm?(APRIL/MAY 2008) (R)**

The efficiency of the backtracking algorithm depends on the following four factors. They are:

   i. The time needed to generate the next xk
   ii. The number of xk satisfying the explicit constraints.
   iii. The time for the bounding functions Bk
   iv. The number of xk satisfying the Bk.

3. What is backtracking?(Nov/Dec 2011) (R)

Backtracking constructs solutions one component at a time and such partially constructed solutions are evaluated as follows ._If a partially constructed solution can be developed further without violating the problem's constraints, it is done by taking the first remaining legitimate option for the next component. ._If there is no legitimate option for the next component, no alternatives for the remaining component need to be considered. In this case, the algorithm backtracks to replace the last component of the partially constructed solution with its next option.

4. What is n-queens problem? (R)

The problem is to place 'n' queens on an n-by-n chessboard so thatno two queens attack each other by being in the same row or in the column orin the same

5. Draw the solution for the 4-queen problem. (R)

	Q		
		Q	
Q			
	Q		

6. Define the Hamiltonian cycle.(NOV/DEC 2012) (R)

The Hamiltonian is defined as a cycle that passes through all the vertices of the graph exactly once. It is named after the Irish mathematician Sir William Rowan Hamilton (1805- 1865).It is a sequence of n+1 adjacentverticesvi0, vi1… vin-1, vi0 where the first vertex of the sequence is same as the last one while all the other n-1 vertices are distinct.

7. What is the subset-sum problem?(Nov/Dec 2012) (R)

Find a subset of a given set S={s1,………,sn} of 'n' positive integers whose sum is equal to a given positive integer 'd'.

8. When can a node be terminated in the subset-sum problem?(NOV/DEC 2008) (R)

The sum of the numbers included are added and given as the value for the root as s'. The node can be terminated as a non-promising node if either of the two equalities holds:
s'+si+1>d (the sum s' is too large)

$$\sum_{j=i\_1}^{n} sj < d \text{ (the sum s' is too small)}$$

9. How can the output of a backtracking algorithm be thought of? (R)

The output of a backtracking algorithm can be thought of as an n-tuple(x1, …xn) where each coordinate xi is an element of some finite linearlyordered set Si. If such a tuple (x1, …xi) is not a solution, the algorithm finds thenext element in Si+1 that is consistent with the values of (x1, …xi) and theproblem's constraints and adds it to the tuple as its (I+1)st coordinate. If suchan element does not exist, the algorithm backtracks to consider the next valueof xi, and so on.

10. Give a template for a generic backtracking algorithm.(APRIL/MAY 2012) (R)
ALGORITHM Backtrack(X [1..i])
　　　　//Gives a template of a generic backtracking algorithm
　　　　//Input X[1..i] specifies the first I promising components of a solution
　　　　//Output All the tuples representing the problem's solution
　　　　if X[1..i] is a solution write X[1..i]
　　　　else
　　　　for each element x Si+1 consistent with X[1..i] and the constraints do
　　　　X[i+1] = x
　　　　Backtrack(X[1..i+1])

11. What is the method used to find the solution in n-queen problem by symmetry? (R)
　　　The board of the n-queens problem has several symmetries so thatsome solutions can be obtained by other reflections. Placements in the lastn/2 columns need not be considered, because any solution with the first queen in square can be obtained by reflection from a solutionwith the first queen in square (1,n-i+1)

12. State m color-ability decision problem.(NOV/DEC 2012) (R)
　　　　Let G be a graph and m be a given positive integer. We want to discover whether thenodes of G can be colored in such a way that no two adjacent nodes have the same color yet only m colors are used.

13. What are the two types of constraints used in Backtracking? (R)
　　✓ Explicit constraints
　　✓ Implicit constraints

14. Define implicit constraint.(APR/MAY 2010& 2012) (R)
　　　　They are rules that determine which of the tuples in the solution space of I satisfy the criteria function. It describes the way in which the xi must relate to each other.

15. Define explicit constraint. (APR/MAY 2010& 2012) (R)
　　　　They are rules that restrict each xi to take on values only from a give set. They depend on the particular instance I of the problem being solved. All tuples that satisfy the explicit constraints define a possible solution space.

16. What is a promising node in the state-space tree? (R) (Nov 17)
　　　　A node in a state-space tree is said to be promising if it corresponds toa partially constructed solution that may still lead to a complete solution.

17. What is a non-promising node in the state-space tree? (R) (Nov 17)
　　　　A node in a state-space tree is said to be promising if it corresponds toa partially constructed solution that may still lead to a complete solution;
otherwise it is called non-promising.

18. What do leaves in the state space tree represent? (R)

Leaves in the state-space tree represent either non-promising deadends or complete solutions found by the algorithm.

19. Define state space tree. (R)

The tree organization of the solution space is referred to as state space tree.

20. Define a live node.(APR/MAY 2010) (R)

A node which has been generated and all of whose children have not yet been generated is called as a live node.

21. Define a dead node. (APR/MAY 2010) (R)

22. Dead node is defined as a generated node, which is to be expanded further all of whose children have been generated.

23. Compared to backtracking and branch and bound?(APRIL/MAY 2008) (AN)

Backtracking	Branch and bound
State-space tree is constructed using depth-first search	State-space tree is constructed using best-first search
Finds solutions for combinatorial non optimizationproblems	Finds solutions for combinatorial optimization problems
No bounds are associated with the nodes in the state-space tree	Bounds are associated with the each and every node in the state-space tree

24. When can a search path are terminated in a branch-and-bound technique. (R)

A search path at the current node in a state-space tree of a branch and-bound algorithm can be terminated if the value of the node's bound is not better than the value of the best solution seen so far the node represents no feasible solution because the constraints of the problem are already violated. The subset of feasible solutions represented by the node consists of asingle point in this case compare the value of the objective function for this feasible solution with thatof the best solution seen so far andupdate the latter with the former if the new solution is better.

25. Give the formula used to find the upper bound for knapsack problem.

A simple way to find the upper bound 'ub' is to add 'v', the total valueof the items already selected, the product of the remaining capacity of theknapsack W-w and the best per unit payoff among the remaining items, whichis v_{i+1}/w_{i+1}, $ub = v + (W-w)(v_{i+1}/w_{i+1})$

26. Define bounding. (R)

✓ Branch-and-bound method searches a state space tree using any search mechanism in which all children of the E-node are generated before another node becomes the E-node.

- ✓ Each answer node x has a cost c(x) and we have to find a minimum-cost answer node. Common strategies include LC, FIFO, and LIFO.
- ✓ Use a cost function ˆc(·) such that ˆc(x) ≤ c(x) provides lower bound on the solution obtainable from any node x.

27. Define Hamiltonian circuit problem. (R)

Hamiltonian circuit problem Determine whether a given graph has a Hamiltonian circuit—

a path that starts and ends at the same vertex and passes through all the other vertices exactly once.

28. State Assignment problem. (MAY\JUNE 2016) (R)

There are n people who need to be assigned to execute n jobs, one person per job. (That is, each person is assigned to exactly one job and each job is assigned to exactly one person.) The cost that would accrue if the ith person is assigned to the jth job is a known quantity [,] for each pair , = 1, 2, . . . , . The problem is to find an assignment with the minimum total cost.

29. What is state space tree? (MAY\JUNE 2016) (R)

Backtracking and branch bound are based on the construction of a state space tree, whose nodes reflect specific choices made for a solution's component .Its root represents an initial state before the search for a solution begins. The nodes of the first level the tree represent the made for the first component of solution, the nodes of the second evel represent the Choices for the second components & so on.

30. Give the purpose of lower bound. (MAY\JUNE 2016) (R)

The elements are compared using operator < to make selection.
Branch and bound is an algorithm design technique that uses lower bound comparisons.
Main purpose is to select the best lower bound.
Example: Assignment problem and transportation problem.

31. What is The Euclidean minimum spanning tree problem? (MAY\JUNE 2016) (R) (Apr 18)

The Euclidean minimum spanning tree or EMST is a minimum spanning tree of a set of n

Points in the plane where the weight of the edge between each pair of points is the Euclidean distance between those two points. In simpler terms, an EMST connects a set of dots using lines such that the total length of all the lines is minimized and any dot can be reached from any other by following the lines.

32. What is an articulation point in the graph? (APR/MAY 2017) (R)

In a graph, a vertex is called an articulation point if removing it and all the edges associated with it results in the increase of the number of connected components in the graph.

33. Write the formula for decision tree for searching a sorted array? (NOV/DEC 2016) (R)

$$C_{worst}(n) \geq \lceil \log_2 n! \rceil. \qquad\qquad (11.2)$$

Using Stirling's formula for $n!$, we get

$$\lceil \log_2 n! \rceil \approx \log_2 \sqrt{2\pi n}(n/e)^n = n \log_2 n - n \log_2 e + \frac{\log_2 n}{2} + \frac{\log_2 2\pi}{2} \approx n \log_2 n.$$

34. State the reason for terminating search path at the current node in branch and bound algorithm. (NOV/DEC 2016) (R)
- The value of the node's bound is not better than the value of the best solution seen so far.
- The node represents no feasible solutions because the constraints of the problem are already violated.
- The subset of feasible solutions represented by the node consists of a single point (and hence no further choices can be made)—in this case we compare the value of the objective function for this feasible solution with that of the best solution seen so far and update the latter with the former if the new solution is better.

35. Differentiate feasible solution and optimal solution. (NOV/DEC 2017)
A *solution* (set of values for the decision variables) for which all of the constraints in the Solver model are satisfied is called a *feasible solution*. An *optimal solution* is a feasible solution where the objective function reaches its maximum (or minimum) value.

36. How is lower bound found by problem reduction? (APR/MAY 2018)

37. What are tractable and non-tractable problems? (APR/MAY 2018)
Generally we think of problems that are solvable by polynomial time algorithms as being tractable, and problems that require super polynomial time as being intractable.

38. Give an example for sum-of-subset problem. (NOV/DEC 2018)
Subset sum problem is to find subset of elements that are selected from a given set whose sum adds up to a given number K. We are considering the set contains non-negative values. It is assumed that the input set is unique (no duplicates are presented).

39. State Hamiltonian circuit problem. (APR/MAY 2019)

PART –B

1. Explain the n-Queen's problem & discuss the possible solutions.(**APRIL/MAY2008**) (**APIRL/MAY2009**)(**NOV/DEC 2008**) (R)

2. Apply backtracking technique to solve the following instance of subset sum problem : S={1,3,4,5} and d=11(**NOV/DEC 2006**) (AN)

3. Explain subset sum problem & discuss the possible solution strategies using backtracking. **(R)**

4. Write a recursive backtracking algorithm to find all the Hamiltonian cycles of a given graph. **(APIRL/MAY2009) (NOV/DEC 2008) (E)**

5. What is backtracking explain detail **(APR/MAY 2007) (R)**

6. Explain graph coloring in detail. **(APIRL/MAY2009) (R)**

7. Give the backtracking algorithm for knapsack problem. **(R)**

8. Explain the algorithm for finding all m-colorings of a graph.**(APRIL/MAY 2012) (R)**

9. Describe the backtracking solution to solve 8-Queens problem**.(APRIL/MAY 2010) (APRIL/MAY2012) (APR/MAY 2017) (A)**

10. With an example, explain Graph Coloring Algorithm.**(APRIL/MAY 2010) (R)**

11. Explain the n-Queen's problem and trace it for n=6.**(APRIL/MAY 2011) (A)**

12.(i)Explain Hamiltonian cycles.   **(NOV/DEC 2012)**   (8) **(R)**

   (ii) With an example, explains Graph Coloring Algorithm   (8) **(R)**
`
13. (i)Explain the control abstraction for Backtracking method. Describe the backtracking solution to solve 8-Queens problem. (8)**(NOV/DEC 2012) (A)**
(ii) Let w= {5, 7, 10, 12, 15, 18, 20} and m=35.Find all possible subset of w whose sum is equivalent to m.Draw the portion of state space tree for this problem. (8) **(A)**

14. How backtracking works on 8-Queens problem with suitable example? **(MAY/JUNE 2013) (A)**

15.(i) Give the backtracking algorithm for knapsack problem. **(MAY/JUNE 2013)**   (8)
   (ii) Explain elaborately recursive backtracking algorithm. (8) **(R)**

 16. Using backtracking, find the optimal solution to a knapsack problem for the knapsack instance n=8,   m=110,(p1…p7)=(11,21,31,33,43,53,55,65)   and (w1…w7)=(1,11,21,33,43,53,55,65). **(A)**

 17. Write an algorithm to determine Hamiltonian cycle in a given graph using back tracking. For the following graph determine the Hamiltonian cycle**. (A)**
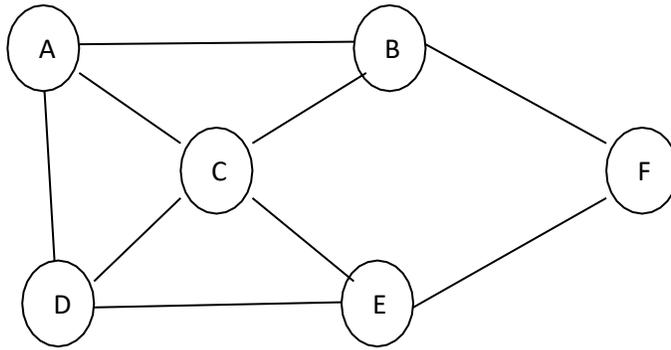
18. Write an algorithm to determine the Sum of Subsets for a given sum and a Set of numbers. Draw the tree representation to solve the subset sum problem given the numbers set as {3, 5, 6, 7, 2} with the sum=15. Derive all the subsets. **(A)**

19. Write an algorithm for N QUEENS problem and Trace it for n=6. **(R)**

20. What is Branch and bound? Explain in detail.**(MAY/JUNE 07) (APIRL/MAY2009) (NOV/DEC 2008) (APR/MAY 2017) (R)**

21. (i)Suggest an approximation algorithm for travelling salesperson problem. Assume that the cost function satisfies the triangle inequality. **(MAY 2015) (R)**

(ii)Explain how job assignment problem could be solved, given n tasks and n agents where each agent has a cost to complete each task, using branch and bound. **(MAY 2015) (AN)**

22. (i)The knight is placed on the first block of an empty board and, moving according to the rules of chess, must visit each square exactly once. Solve the above problem using backtracking procedure. **(MAY 2015) (AN)**

(ii)Implement an algorithm for Knapsack problem using NP-Hard approach. **(MAY 2015) (R)**

23. State the subset-sum problem and Complete state-space tree of the backtracking algorithm applied to the instance A={3, 5, 6, 7} and d=15 of the subset-sum problem.**(MAY\JUNE 2016) (A)**

24. (i) Draw a decision tree and find the number of key comparisons in the worst and average cases for the three element bubble sort. (8) **(NOV/DEC 2016) (AN)**

(ii) Write backtracking algorithm for 4 queen's problem and discuss the possible solution. (8) **(NOV/DEC 2016) (R)**

25. Solve the following instance of knapsack problem by branch and bound algorithm W= 15.(16) **(NOV/DEC 2016) (AN)**

ITEMS	WEIGHT	PROFIT
1	5	40
2	7	35
3	2	18
4	4	4
5	5	10
6	1	2

26. Apply Branch and bound algorithm to solve the travelling salesman problem. **(NOV/DEC 2016) (A)**
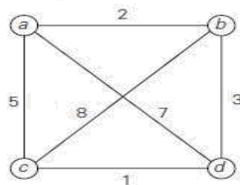
27. Give the methods for Establishing Lower Bound. **(NOV/DEC 17)**

28. Find the optimal solution using branch and bound for the following assignment problem.

	Job1	Job 2	Job 3	Job 4
A	9	2	7	8
B	6	4	3	7
C	5	8	1	8
D	7	6	9	4

29. Elaborate on the nearest-neighbor algorithm and multifragment- heuristic algorithm for TSP problem. **(APR/MAY 2018)**

30. Consider the travelling salesperson instances defined by the following cost matrix. **(NOV/DEC 2018)**

$$
\begin{array}{ccccc}
\infty & 20 & 30 & 10 & 11 \\
15 & \infty & 16 & 4 & 2 \\
3 & 5 & \infty & 2 & 4 \\
19 & 6 & 18 & \infty & 3 \\
16 & 4 & 7 & 16 & \infty
\end{array}
$$

Draw the state space and show the reduced matrices corresponding to each of the node. (13)**(NOV/DEC 2018)**

31. Write an algorithm for subset sum and explain with an example. (13)**(APR/MAY 2019)**

32. Explain the 4-queens problem using backtracking. Write the algorithms . Giv the estimated cost for all possible solutions of 4-queens problem.Specify the implicit and explicit constrints. **(APR/MAY 2019)**

UNIT V

NP-COMPLETE AND APPROXIMATION ALGORITHM

Tractable and intractable problems: Polynomial time algorithms – Venn diagram representation - NP- algorithms - NP-hardness and NP-completeness – Bin Packing problem - Problem reduction: TSP – 3-CNF problem. **Approximation Algorithms**: TSP - **Randomized Algorithms**: concept and application - primality testing - randomized quick sort - Finding k^{th} smallest number

1. **What are NP- hard and NP-complete problems? (R)**
The problems whose solutions have computing times are bounded by polynomials of small degree.

2. **Define bounding. (R)**
   ✓ Branch-and-bound method searches a state space tree using any search mechanism in which all children of the E-node are generated before another node becomes the E-node.
   ✓ Each answer node x has a cost c(x) and we have to find a minimum-cost answer node. Common strategies include LC, FIFO, and LIFO.
   ✓ Use a cost function ˆc(·) such that ˆc(x) ≤ c(x) provides lower bound on the solution obtainable from any node x.

3. **List example of NP hard problem. (R)**
   NP hard graph problem
   ✓ clique decision problem(CDP)
   ✓ Node cover decision problem(NCDP).

4. **What is meant by NP hard and NP complete problem?(NOV/DEC 2011&NOV/DEC 2012) (R)**
   ✓ NP-Hard Problem: A problem L is NP-hard if any only if satisfy ability reduces to L.
   ✓ NP- Complete: A problem L is NP-complete if and only if L is NP-hard and L є NP.
   There are NP-hard problems that are not NP-complete.
   Halting problem is NP-hard decision problem, but it is not NP-complete.

5. **An NP-hard problem can be solved in deterministic polynomial time,how?(NOV/DEC 2012)**

- ✓ If there is a polynomial algorithm for any NP-hard problem, then there are polynomial algorithms for all problems in NP, and hence P = NP.
- ✓ If P ≠ NP, then NP-hard problems cannot be solved in polynomial time, while P = NP does not resolve whether the NP-hard problems can be solved in polynomial time.

6. How NP-Hard problems are different from NP-Complete? (R)

These are the problems that are even harder than the NP-complete problems. Note that NP-hard problems do not have to be in NP, and they do not have to be decision problems.

The precise definition here is that a problem X is NP-hard, if there is an NP- complete problem Y, such that Y is reducible to X in polynomial time.

But since any NP-complete problem can be reduced to any other NP-complete problem in polynomial time, all NP-complete problems can be reduced to any NP-hard problem in polynomial time. Then, if there is a solution to one NP-hard problem in polynomial time, there is a solution to all NP problems in polynomial time.

7. Define P and NP problems. (APR/MAY 2017) (R)

P- Polynomial time **solving** . Problems which can be solved in polynomial time, which take time like O(n), O(n2), O(n3). Eg: finding maximum element in an array or to check whether a string is palindrome or not. so there are many problems which can be solved in polynomial time.

NP- Non deterministic Polynomial time solving. Problem which can't be solved in polynomial time like TSP(travelling salesman problem)

8. What are tractable and non-tractable problems? (APR/MAY 2018)

Generally we think of problems that are solvable by polynomial time algorithms as being tractable, and problems that require super polynomial time as being intractable.

9. Define P and NP problems. (NOV/DEC 2018)

All problems in P can be solved with polynomial time algorithms, whereas all problems in NP - P are intractable.

It is not known whether P = NP. However, many problems are known in NP with the property that if they belong to P, then it can be proved that P = NP.

If P ≠ NP, there are problems in NP that are neither in P nor in NP-Complete.

The problem belongs to class P if it's easy to find a solution for the problem. The problem belongs to NP, if it's easy to check a solution that may have been very tedious to find.

10. Define NP completeness and NP hard.(APR/MAY 2019)
NP-complete problems are the hardest problems in NP set. A decision problem L is NP-complete if:
1) L is in NP (Any given solution for NP-complete problems can be verified quickly, but

there is no efficient known solution).
2) Every problem in NP is reducible to L in polynomial time (Reduction is defined below).
A problem is NP-Hard if it follows property 2 mentioned above, doesn't need to follow
property 1. Therefore, NP-Complete set is also a subset of NP-Hard set.

11. What do you meant by primality testing?
The basic structure of randomized primality tests is as follows:
- Randomly pick a number a.
- Check equality (corresponding to the chosen test) involving a and the given
  number n. If the equality fails to hold true, then n is a composite number and a is
  a witness for the compositeness, and the test stops.
- Get back to the step one until the required accuracy is reached.
After one or more iterations, if n is not found to be a composite number, then it can be
declared probably prime.

12. What is Kth smallest number?
Given an array and a number k where k is smaller than the size of the array, we need to find
the k'th smallest element in the given array. It is given that all array elements are distinct.

13. How quick sort using random pivoting?
In QuickSort we first partition the array in place such that all elements to the left of the pivot
element are smaller, while all elements to the right of the pivot are greater than the pivot.
Then we recursively call the same procedure for left and right subarrays.
Unlike merge sort, we don't need to merge the two sorted arrays. Thus Quicksort requires
lesser auxiliary space than Merge Sort, which is why it is often preferred to Merge Sort.
Using a randomly generated pivot we can further improve the time complexity of QuickSort.

PART –B

1. Suggest an approximation algorithm for travelling salesperson problem. Assume that the
   cost function satisfies the triangle inequality. **(MAY 2015) (R)**

2. Implement an algorithm for Knapsack problem using NP-Hard approach. **(MAY 2015) (R)**

3. Discuss the approximation algorithm for NP hard problem? **(NOV/DEC 2016) (R)**

4. What is class NP? Discuss about any five problems for which no polynomial time for TSP
   problem. **(APR/MAY 2018)**

5. Elaborate on the nearest-neighbor algorithm and multifragment- heuristic algorithm for TSP
   problem. **(APR/MAY 2018)**

6. Discuss the approximation algorithm for NP-hard problem. (13)**(NOV/DEC 2018)**

7. Write an algorithm to solve the travelling salesman problem and prove that it is a 2 time approximation algorithm.(13)**(APR/MAY 2019)**

COURSE OUTCOMES:

Course Name   : CS3401 - DESIGN AND ANALYSIS OF ALGORITHMS

Year/Semester   : II/ IV

Year of Study   : 2022 –2023 (R – 2021)

On Completion of this course student will be able to

COURSE OUTCOMES

CO	DESCRIPTION
CO1	Analyze the efficiency of algorithms using various frameworks
CO2	Apply graph algorithms to solve problems and analyze their efficiency.
CO3	Make use of algorithm design techniques like divide and conquer and dynamic programming
CO4	Use design techniques like greedy technique to solve a problem.
CO5	Use the state space tree method for solving problems
CO6	Solve problems using approximation algorithms and randomized algorithms

CO-PO MATRIX:

CO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
CO.1	2	1	3	2	-	-	-	-	2	1	2	3
CO.2	2	1	1	1	1	-	-	-	1	3	3	3
CO.3	1	3	3	3	1	-	-	-	1	2	1	2
CO.4	1	3	3	3	1	-	-	-	1	2	1	2
CO.5	1	2	2	3	-	-	-	-	2	3	3	1
CO.6	1	2	3	2	3	-	-	-	3	1	3	3
CO	1	2	2	2	2	-	-	-	2	2	2	2

CO – PSO MATRIX:

CO	PSO1	PSO2	PSO3
CO.1	2	1	1
CO.2	2	3	3
CO.3	2	1	2
CO.4	2	1	2
CO.5	3	1	3
CO.6	1	3	3
CO	2	2	2